3. البرامج المرتبطة بالمترجمات

Programs to Compilers Related

سيتم في هذا الجزء إعطاء وصف مختصر لمجموعة البرامج المرتبطة بالمترجمات أو التي تستخدم معها والتي غالبا ما تأتي معها ضمن البيئة الكاملة الخاصة بإعداد البرامج بلغة البرمجة حيث تشمل هذه المجموعة البرامج الآتية:

1.3. المترجمات الفورية Interpreters

هى ايضا مترجمات للغات البرمجة مثل باقى المترجمات ولكنها تختلف في كونها تقوم بتنفيذ برنامج المصدر (Source Program) مباشرة دون توليد شفرة الهدف التي يتم تنفيذها بعد الانتهاء من الترجمة بالكامل كما يحدث في حالة استخدام المترجمات المعتادة (Compilers) ويفضل استخدام المترجمات الفورية مع بعض لغات البرمجة مثل لغة ال BASIC وخاصة في الأغراض التعليمية وذلك لكونها تقوم بتوضيح الاخطاء الموجودة في البرنامج خطوة خطوة وأولا بأول ولكن يعاب على هذا النوع من المترجمات أنها تقوم بإعادة الترجمة في كل مرة يتم فيها تنفيذ البرنامج لذلك فإن المترجمات المعتادة يفضل استخدامها إذا كانت السرعة مطلوبة في تنفيذ البرنامج وذلك لأن تنفيذ النسخة المترجمة من البرنامج يكون أسرع من تنفيذ البرنامج المصدر بدرجة تصل إلى عشرة أضعاف وعموماً فإن كلاً من المترجمات الفورية والمترجمات المعتادة يتشابهان ويشتركان في عدد كبير من العمليات ولكننا سنركز في شرحها في تلك المادة العلمية على المترجمات المعتادة فقط.

2.3 المجمعات Assemblers

المجمّع هو مترجم للغة تجميع خاصة بحاسب محدد وكما ذكرنا سابقا فإن لغة التجميع هو شكل رمزى للغة الآلة الخاصة بتلك الحاسب ولذلك فإنها أسهل في الترجمة وفي بعض الاحيان تقوم المترجمات المعتادة الخاصة بلغات البرمجة بتوليد لغة التجميع المقابلة لتلك اللغات ثم يقوم المجمع بتحويلها إلى لغة الآلة.

3.3 برامج الربط Linkers

تحتاج كل من المترجمات والمجمعات في الغالب إلى برنامج يسمى الرابط الذي يقوم بدمج شفرات الملفات الناتجة عن عملية الترجمة أو التجميع في ملف مستهدف واحد قابل للتنفيذ مباشرة وايضا يقوم الرابط باضافة شفرة الوظائف الجاهزة (Built-in Functions) المستخدمة وبعض الخدمات المطلوبة من نظام التشغيل الخاص بالحاسب إلى ذلك الملف التنفيذى المستهدف وعموما فإن الدور الذي يقوم به الرابط كان في السابق ضمن وظائف المترجمات ولكن تم فصله بعد ذلك ولهذا فلن يتم شرحه في هذه المادة العلمية.

4.3 برامج التحميل Loaders

غالبا ما تقوم المترجمات والمجمعات بتوليد شفرة البرنامج المستهدف دون أن تحتوى على تحديد للمواقع التخزينية التي في الذاكرة بشكل مطلق بل أنها تتوقف على موقع البرنامج نفسه في الذاكرة ويتم ذلك لاعطاء الفرصة لتحميل البرنامج في أي موقع بالذاكرة بالاضافة إلى عدم الاحتياج إلى إعادة الترجمة عند نقله من موقع لآخر حيث يقوم برنامج التحميل بتحويل تلك المواقع التخزينية القابلة للنقل إلى مواقع تخزينية ثابتة عند قيامه بتحميل البرنامج إلى الذاكرة.

5.3 برامج التحرير Editors

إن المترجمات دائما ما تقبل برامج المصدر (Source Programs) المكتوبة باستخدام أحد برامج التحرير لهذا فإن عملية الترجمة غالباً ما تتم من داخل تلك البرامج لكى تكوّن بيئة تفاعلية متكاملة لتحرير البرامج وفي هذه الحالة يكون برنامج التحرير مرتبط بلغة محددة من لغات البرمجة وهى اللغة التي يتضمن البرنامج المترجم الخاص بها وعند ذلك يكون برنامج التحرير موجه لتلك اللغة ومهيكل وفقاً للصيغ النحوية الخاصة بها مما يسهل من عملية تحرير البرامج بهذه اللغة.

4. مراحل المترجم Compiler Stages

يتكون المترجم من مجموعة من الخطوات أو المراحل التي تقوم بانجاز عدد من العمليات المختلفة ومن الأفضل أن نفكر في تلك المراحل على أنها أجزاءً منفصلة داخل المترجم على الرغم من كونها من الناحية العملية يكونوا وحدة واحدة والشكل التالي يوضح مراحل المترجم المختلفة بالاضافة إلى ثلاثة مكونات اضافية تتفاعل مع جزء أو كل هذه المراحل:

وسنقوم بوصف تلك المراحل باختصار في هذا الجزء على أن يتم دراسة المراحل الثلاثة الأولى بالتفصيل في الوحدات الرابعة والخامسة والسادسة على التوالي وبالنسبة لمراحل المترجم الأخرى فلن يتم شرحها في هذه المادة العلمية. أما المكونات الثلاثة الإضافية فسيتم عرضها في الجزء الأخير من هذه الوحدة.

1.4 محلل المفردات Lexical Analyzer

إن محلل المفردات أو مايطلق علية أحيانا الماسح (Scanner) هو الذي يقوم بالقراءة الفعلية لبرنامج المصدر على هيئة سلسلة متتابعة من الحروف حيث يقوم بتجميع سلاسل الحروف التي تكون وحدات ذات معنى تكون مفردات (Tokens) لغة البرمجة وبالتالي فإن دور محلل المفردات هو تمييز مفردات اللغة فعلى سبيل المثال بفرض سطر الشفرة التالي المكتوب بلغة السى:

a [index] = 4 + 2

هذا السطر يتكون من 12 حرفاً وبحذف المسافات تكون 8 مفردات كالاتى:

a معرف identifier

[left bracket (قوس ايسر (فتحة قوس

index identifier معرف

] right bracket (قوس ايمن (اغلاق قوس

= assignment sign رمز التخصيص

4 number عدد

+ plus sign رمز الجمع

2 number عدد

وكل مفردة من المفردات السابقة تتكون من حرف واحد أو أكثر يتم تجميعها في وحدة واحدة قبل القيام بالمراحل التالية ومحلل المفردات قد يقوم بعمليات أخرى بالإضافة إلى تمييز المفردات مثل إضافة المعرفات إلى جدول الرموز وإضافة الثوابت إلى جدول الثوابت.

2.4. محلل الصيغ النحوية Syntax Analyzer

إن مرحلة تحليل الصيغ النحوية أو ما يطلق عليها أحياناً مرحلة الإعراب (Parsing) تتلقى شفرة المصدر في شكل مفردات من محلل المفردات ليقوم بتحليل الصيغ النحوية لتحديد هيكل البرنامج وذلك مثل تحليل القواعد لجملة من جمل اللغات الطبيعية ونتيجة لهذه المرحلة يتم إنشاء شجرة الإعراب (Parse Tree) أو شجرة النحو (Syntax Tree) وكمثال على ذلك بفرض سطر الشفرة السابق المكتوب بلغة السى فإنه عبارة عن تعبير (Expression) يمثل الهيكل الخاص بأمر التخصيص في تلك اللغة وهذا الهيكل يمكن تمثيلة باستخدام شجرة الإعراب التالية:

الشكل (2)

حيث يلاحظ أن العقد الداخلية من شجرة الإعراب معنونة بأسماء التراكيب التي يمثلها بينما أوراق شجرة الإعراب تمثل تتابع المفردات من المدخلات.

وأحيانا يتم انشاء شجرة النحو بدلا من شجرة الإعراب حيث تختصر شجرة النحو بعض المعلومات المعروضة في شجرة الإعراب لانها أكثر منها تجريداً. فشجرة النحو الخاصة بمثال أمر التخصيص السابق تكون كما يلى:

ويلاحظ أن عدد من العقد لم تظهر في شجرة النحو بما فيها بعض المفردات.

3.4. محلل الدلالات Semantic Analyzer

إن دلالات البرنامج هى المعنى الخاص في مقابلة صيغه النحوية ويتم تحديد دلالات أي برنامج من خلال سلوكه أثناء التشغيل ولكن معظم لغات البرمجة لديها بعض الصفات التي يمكن تحديدها قبل البدء في التشغيل ولكن تلك الصفات لا يستطيع وصفها أو تحليلها بواسطة محلل الصيغ النحوية ويطلق على هذه الصفات لقب الدلالات الثابتة (Static Semantics) ومهمة محلل الدلالات هو تحليل هذا النوع من الدلالات التي تشمل بشكل أساسي التعريفات (Declarations) واختبار الأنواع (Type Checking) والمعلومات الإضافية التي تعكس هذه الدلالات تسمى خصائص (Attributes) ويتم تحديدها أثناء المرحلة الحالية وغالباً ما تتم إضافتها كحواشي لشجرة الإعراب أو شجرة النحو وقد تضاف أيضا لجدول الرموز. ففي مثال أمر التخصيص السابق نجد أن المعلومات الاساسية الخاصة بالنوع والتي يجب تجميعها قبل البدء في تحليل الدلالات الخاصة بهذا الأمر هي ان المتغير a عبارة عن مصفوفة للقيم الصحيحة (Integers) وأن مدى الفهرس الخاص بتلك المصفوفة يجب أن يكون من القيم الصحيحة ايضا وبالفعل المتغير Index هو متغير صحيح ثم يقوم محلل الدلالات باضافة تلك الخصائص كحواشي لشجرة النحو لتصبح كما يلي:

وبعد الانتهاء من ذلك يتم التأكد من أن أمر التخصيص يمكن أن يتم بالنسبة لهذا النوع من البيانات فإذا كان ذلك صحيح كما في المثال فإن محلل الدلالات يعلن توافق أنواع البيانات في الأمر وإلا فانه يعطى رسالة خطأ تفيد عدم وجود هذا التوافق.

4.4. محسن شفرة المصدر Source Code Optimizer

إن المترجمات عادة ما تشتمل على عدد من الخطوات الخاصة بتحسين شفرة البرنامج وفي أغلب الأحوال تكون الخطوة الأولى منها بعد الانتهاء من تحليل الدلالات حيث يكون هناك إمكانية لمثل هذا التحسين ولكنه مرتبط فقط بشفرة المصدر الذي يظهر كمرحلة منفصلة من مراحل الترجمة و عموماً فإن المترجمات المتعددة تختلف فيما بينها ليس فقط في نوع التحسين الذي يتم ولكن أيضا في موقع ذلك التحسين ضمن مراحل الترجمة.

ففي مثالنا السابق هناك فرصة لهذا التحسين على مستوى برنامج المصدر يتمثل في التعبير 2 + 4 حيث يمكن حسابه في هذه المرحلة واستبداله بالناتج 6 وهذا التحسين يمكن أن يتم مباشرة في شجرة النحو ذات الحواشي عن طريق دمج الجانب الأيمن من الشجرة ليعبر فقط عن قيمة ثابتة كما يلي :

ويوجد تحسينات متعددة يمكن إجراءها مباشرة على الشجرة ولكن في أغلب الحالات يكون ذلك أسهل تنفيذه على الشكل الخطي للشجرة الذي يكون قريباً من شفرة لغة التجميع وهي ما تسمى بالشفرة الوسيطة (Intermediate Code) وذلك في إشارة إلى كون هذه الشفرة تقع بين شفرة المصدر وشفرة الهدف والتي تعتبر تمثيل داخلي لبرنامج المصدر يستخدم فقط بواسطة المترجم وعلى هذا فإن شجرة النحو يمكن أيضاً اعتبارها شفرة وسيطة ولذلك فإن الشفرة الوسيطة يشار إليها في بعض الأحيان بكونها تمثيل وسيط (Intermediate Representation) وليس مجرد شفرة.

5.4 مولد الشفرة Code Generator

مولد الشفرة يأخذ كمدخلات الشفرة أو التمثيل الوسيط ليقوم بتوليد شفرة خاصة بالآلة المستهدفة والتي هي شفرة الهدف وفي هذه المرحلة من الترجمة فإن خصائص الآلة (الحاسب) المستهدفة يكون لها التأثير الأساسي وهذا ليس فقط من حيث ضرورة استخدام الأوامر الموجودة في الآلة المستهدفة ولكن أيضا عند اتخاذ القرارات الخاصة بتمثيل البيانات والتي يتم فيها تحديد المساحة التي يشغلها كل بيان في الذاكرة .

6.4. محسّن شفرة الهدف Object Code Optimizer

في هذه المرحلة يحاول المترجم إجراء التحسينات في شفرة الهدف التي تم توليدها في المرحلة السابقة وهذه التحسينات تشمل اختيار أسلوب العنونة (Addressing Mode) المناسب الذي يزيد من سرعة تنفيذ البرنامج واستبدال الأوامر البطيئة بأوامر أسرع وكذلك حذف العمليات المتكررة غير الضرورية.

وبنهاية هذه المرحلة ينتهي الوصف المختصر لمراحل المترجم مع ضرورة التأكيد أن هذا الوصف يوضح فقط وظيفة كل مرحلة وليس من الضروري أن يعكس التنظيم الفعلي للمترجمات والتي تختلف كثيراً عن بعضها البعض في تفاصيلها التنظيمية ولكن مع ذلك فإن المراحل التي تم استعراضها موجودة في أغلب المترجمات. هذا وقد تمت الإشارة في هذا الجزء لعدد من تراكيب البيانات (Data Structures) اللازمة للاحتفاظ بالمعلومات التي تحتاجها كل مرحلة من مراحل المترجم مثل شجرة النحو والشفرة الوسيطة وجدول الثوابت وأيضاً جدول الرموز والتي سنخصص الجزء التالي لتقديم عرض مبسط لتراكيب البيانات الأساسية التي يستخدمها المترجم.

5. تراكيب البيانات الأساسية في المترجم

Types of Data Structure in Compilers

إن العلاقة بين الخوارزميات المستخدمة بواسطة مراحل المترجم وبين تراكيب البيانات التي تدعم تلك المراحل علاقة قوية فالذي يقوم بإعداد المترجم يحاول أن يطبق هذه الخوارزميات بأكفأ الأساليب دون أن يؤدي ذلك إلى زيادة التعقيد بقدر كبير وهو ما يتم التعبير عنه بأن المترجم يجب أن يكون لديه القدرة على ترجمة البرنامج في الوقت الذي يتناسب مع حجمه وفي هذا الجزء سنقوم بتقديم تراكيب البيانات الرئيسية التي يحتاج إليها المترجم في مراحله المختلفة والتي تعتبر جزءً أساسيا من عملياتها وتخدم في نقل المعلومات بينها.

1.5. المفردات Tokens

عندما يقوم محلل المفردات (Lexical Analyzer) بتجميع الحروف التي تكون مفردة محددة فانه بصفة عامة يتم تمثيل هذه المفردة في شكل قيمة من القيم المحددة التي تمثل مجموعة المفردات الخاصة بلغة المصدر وفي بعض الأوقات يكون من الضروري الاحتفاظ بسلسلة الحروف التي تمثل المفردة وكذلك بعض المعلومات المشتقة منها مثل الاسم الخاص بالمعرف أو قيمة المفردة الرقمية وفي معظم لغات البرمجة يحتاج محلل المفردات إلى توليد مفردة واحدة فقط في كل مرة وفي هذه الحالة فإن متغير واحد عام يمكن استخدامه للاحتفاظ بمعلومات المفردة وفي حالات أخرى مثل لغة الفورتران تستخدم مصفوفة للاحتفاظ بمعلومات عن المفردات.

2.5. شجرة النحو Syntax Tree

إن محلل الصيغ النحوية يقوم بتوليد شجرة النحو والتي عادة ما يتم إنشاؤها كتركيبة للبيانات قائمة على استخدام المؤشرات (Pointers) حيث يتم بناءها بشكل مرحلي وفقاً لتقدم عملية الإعراب الخاصة بشفرة المصدر والشجرة بالكامل يتم الاحتفاظ بها باستخدام متغير واحد يشير إلى جذر الشجرة وكل عقدة (Node) في الشجرة عبارة عن سجل (Record) تمثل الحقول (Fields) الخاصة بالمعلومات التي يتم تجميعها أثناء مرحلة تحليل الصيغ النحوية أو مرحلة تحليل الدلالات فمثلا نوع بيانات تعبير رياضي يتم الاحتفاظ به كحقل في داخل عقدة شجرة النحو الخاصة بذلك التعبير وعموماً فإن كل عقدة في شجرة النحو قد تطلب خصائص مختلفة لتخزينها وفقاً لنوع تركيبة لغة البرمجة التي تمثله فالعقدة التي تمثل تعبير رياضي تحتفظ بخصائص تختلف عن الخصائص التي تحتفظ بها العقدة التي تمثل أحد الأوامر ولهذا فإن عقد شجرة النحو تختلف في حقولها عن بعضها البعض.

3.5. جدول الرموز Symbol Table

هو تركيبة البيانات التي تحتفظ بمعلومات ترتبط بالمعرفات المختلفة من وظائف (Functions) ومتغيرات (Variables) وثوابت (Constants) وأنواع بيانات (Data Types) وهذا الجدول يتفاعل مع جميع مراحل المترجم حيث يتم إنشاء الخانة الخاصة بالمعرف في الجدول أثناء مرحلة تحليل المفردات أو مرحلة تحليل الصيغ النحوية ثم تقوم مرحلة تحليل الدلالات بإضافة نوع المعرف وباقي المعلومات الخاصة بها على أن يتم استخدام تلك المعلومات أثناء مراحل توليد وتحسين شفرة الهدف ولكون جدول الرموز يتم استخدامه بكثرة خلال عملية الترجمة سواء في الإضافة أو الحذف أو الاسترجاع لذلك فإن كفاءة القيام بهذه المهام تعتبر من الأمور الهامة من أجل كفاءة عملية الترجمة بصفة عامة ولهذا فعادة ما تستخدم الجداول العشوائية (Hash Tables) كتركيبة بيانات أساسية لتمثيل جدول الرموز لما تتميز بها من كفاءة وسرعة عالية في الإضافة والحذف واسترجاع المعلومات .

4.5. جدول الثوابت Literal Table

إن هذا الجدول يقوم بتخزين الثوابت الحرفية والرقمية المستخدمة داخل البرنامج ولا يحتاج هذا الجدول للقيام بحذف للثوابت التي يتضمنها وذلك لكون تلك الثوابت تكون لها صفة العمومية داخل البرنامج ككل وليست مرتبطة بجزء محدد من أجزاء البرنامج كما أن كل منها يظهر لمرة واحدة فقط داخل الجدول وعموماً فإن جدول الثوابت له دور مهم في تخفيض الحجم الذي يشغله البرنامج داخل الذاكرة كما أنه يستخدم أثناء مرحلة توليد شفرة الهدف.

5.5. الشفرة الوسيطة Intermediate Code

حسب نوع الشفرة الوسيطة التي يتم استخدامها ووفقاً لنوع التحسين الذي يتم إنجازه فإن هذه الشفرة الوسيطة قد يتم الاحتفاظ بها داخل مصفوفة من سلاسل الحروف أو في ملف نصي (Text File) مؤقت وعموماً فإن المترجمات التي تقوم بعمليات تحسين معقدة لشفرة البرنامج تعطي أهمية كبيرة لاختيار التمثيل الأفضل للشفرة الوسيطة الذي يسهل من عملية التحسين.

6.5. الملفات المؤقتة Temporary Files

إن الحواسيب في السابق لم يكن لديها حجم الذاكرة الكافي للاحتفاظ بالبرنامج كاملاً في الذاكرة أثناء الترجمة ولحل هذه المشكلة فإنه غالباً ما كان يتم اللجوء لاستخدام مجموعة من الملفات المؤقتة للاحتفاظ بالمخرجات الوسيطة لكل مرحلة من مراحل الترجمة وعموماً فإن القيد الخاص بحجم الذاكرة المحدود أصبح غير موجود في هذه الأيام أو على الأقل يمثل مشكلة بسيطة لهذا أصبح من الممكن القيام بالترجمة بالكامل داخل الذاكرة دون الاحتياج إلى أي ملفات مؤقتة وخاصة في حالة إمكانية تجزئة برنامج المصدر و القيام بترجمة كل جزء على حده .
لغة البرمجة الافتراضية

Sample Programming Language

إن إعداد مادة علمية خاصة بالمترجمات لا تصبح كاملة بدون إعطاء أمثلة لكل خطوة من خطوات عملية الترجمة وفي حالات كثيرة سوف نوضح الأساليب التي نتعرض لها في هذه المادة العلمية باستخدام أمثلة من لغات البرمجة القائمة والمعروفة مثل لغة الـ C والـ C++ والـ Pascal والـ Ada ولكن هذه الأمثلة تكون غير كافية لعرض كيفية تجميع أجزاء المترجم الخاص بأحد لغات البرمجة واستيعابها. وبما أن القيام بذلك الأمر يكون من الصعوبة بالنسبة للغات البرمجة الكاملة لهذا فإن الحل البديل هو استخدام لغة برمجة افتراضية لتوضيح ذلك وعموماً فإن اللغة الافتراضية المستخدمة في هذه المادة العلمية هي لغة برمجة بسيطة وسوف تستخدم في الوحدات الثلاثة الأخيرة من تلك المادة العلمية لتطبيق المفاهيم التي سيتم عرضها.

وهيكل البرنامج المكتوب بهذه اللغة الافتراضية يكون بسيطاً لكونه يتكون فقط من مجموعة من الأوامر يفصل بينهم الفاصلة المنقوطة باستخدام صيغ نحوية مشابهة للصيغ النحوية الخاصة بلغة الـ Pascal وفي تلك اللغة الافتراضية لا توجد إجراءات (Procedures) أو حتى تعريفات (Declarations) وجميع المتغيرات متغيرات صحيحة (Integer Variables) ولا يتم تعريفها مقدماً بل تأخذ القيم الخاصة بها مباشرة كما يحدث في لغة الـ Basic ويوجد بهذه اللغة أمران مركبان فقط هما أمر if

الشرطي و أمر repeat التكراري وكلا الأمرين يمكن أن يتضمنا عدة أوامر وينتهي أمر if بكلمة end ولدى اللغة الافتراضية أيضاً الأمران read و write للقيام باستقبال المدخلات وإعطاء المخرجات على التوالي وبالنسبة للتعبيرات الرياضية في هذه اللغة فهي محدودة ومقصورة على التعبيرات الصحيحة أو المنطقية والتعبيرات المنطقية تتكون من مقارنة بين تعبيرين رياضيين باستخدام عمليتي المقارنة '<' و '=' فقط أما التعبيرات الصحيحة فتشمل الثوابت والمتغيرات الصحيحة والأقواس والعمليات الحسابية الأربعة الأساسية '+' و '-' و '*' و '/' وأخيراً فإن تلك اللغة تسمح بإضافة التعليقات (Comments) إلى البرنامج على أن تكون محصورة بين الأقواس { } .

والبرنامج التالي هو برنامج بسيط مكتوب بلغة البرمجة الافتراضية للقيام بحساب المضروب (Factorial) وهذا البرنامج سوف يستخدم كمثال عبر تلك المادة العلمية:

```
{Sample program to compute factorial}
;read x
if x > 1 then
;fact : = 1
repeat
; fact : = fact * x
; x : = x - 1
; until x = o
write fact
end
```

الطرق المعتادة لوصف النحو (الصيغ النحوية)

في هذا الجزء سيتم استعراض أهم الطرق التي تستخدم لوصف الصيغ النحوية (Syntax) الخاصة بلغات برمجة الحاسوب والتي يعتمد عليها في عملية الترجمة من لغة المصدر إلى لغة الهدف وذلك لأن ترجمة أي برنامج مكتوب بإحدى لغات البرمجة تشمل التأكد من أن هذا البرنامج متوافق مع الصيغ النحوية الخاصة بلغة البرمجة المكتوب بها.

1.1. القواعد خالية السياق Context-free Grammars

هذه القواعد ظهرت في منتصف الخمسينات من القرن الماضي على يد عالم اللغويات كومسكاي (Chomsky) الذي قام باستخدامها في وصف قواعد اللغات الطبيعية (Natural Languages) ولكن سرعان ما بدأ إستخدام تلك القواعد في وصف الصيغ النحوية للغات البرمجة حيث ظهر نجاحها بعد

ذلك إلى حد بعيد وبصفة عامه تعتبر هذه القواعد هي الأساس الذي قامت عليه جميع الأشكال التي ظهرت بعد ذلك لوصف الصيغ النحوية للغات البرمجة.

2.1. شكل باكوس نور Backus Naur Form

بعد كومسكاي قام جون باكوس (John Backus) في أواخر الخمسينات من القرن الماضي باستخدام شكل جديد لوضع الصيغ النحوية للغة الألجول 58 (Algal58) التي تعتبر من أوائل لغات برمجة الحاسوب. هذا الشكل تم إدخال بعض التعديلات عليه على يد بيتر نور (Peter Naur) ، وتم استخدام الشكل المعدل في وصف لغة الألجول 60 (Algal 60) حيث أصبح هذا الشكل بعد التعديل معروف باسم شكل باكوس نور Backus Naur Form) ، أو كما يطلق عليه اختصاراً (BNF) ، وتم استخدامه في وصف الصيغ النحوية لأغلب لغات برمجة الحاسوب التي ظهرت بعد ذلك.

شكل باكوس نور يعطي أسماء مجردة (Abstraction) للتراكيب النحوية التي تتكون منها لغة البرمجة فعلى سبيل المثال لوصف أمر التخصيص في لغة الجافا (Java) يتم ذلك كما يلي:

< assign >   <var> = < expression >

والرمز الموجود على الجانب الأيسر من السهم ( ) هو الاسم المجرد للتركيب النحوي المراد تعريفه وهو أمر التخصيص في هذه الحالة والمشار له بالإسم المجرد <assign> أما باقي التعريف والذي يظهر على الجانب الأيمن للسهم ( ) فإنه يتكون في هذا المثال من مجموعة من الرموز (مثل إشارة التخصص (=)) والأسماء المجردة الأخرى (<expression> & <var>) والتي يجب أن يتم توصيف كلا منها على حده في تعريف منفصل. وخلاصة التعريف السابق هو أن أمر التخصيص في لغة الجافا يتكون من متغير (< var>) وتعبير رياضي أو تعبير نصي (< expression>) وبينهما إشارة التخصيص (=) وكما هو واضح من التعريف فإن المتغير لابد أن يكون على يسار رمز التخصيص بينما التعبير النصي أو الرياضي فإنه يكون على يمين رمز التخصيص ولكي يكتمل وصف أمر التخصيص لابد أن يتم توصيف كل من المتغيرات (<var>) وكذلك التعبيرات (<expression) في تعريف منفصل لكلا منهم.

والأمر التالي يوضح مثال لأمر التخصيص في لغة الجافا وفقا للتوصيف الموضح بالتعريف السابق باستخدام شكل الـ شكل باكوس نور

Total = Subtotal1 + Subtotal2

وعموما فإن الأسماء المجردة (Abstraction) يطلق عليها رموزاً غير نهائية (Non- terminal Symbols) وذلك لأنها تحتاج إلى توصيفها بعد ذلك بينما الرموز فيطلق عليها (Terminal Symbols) أي رموز نهائية لأن تعريفها منتهٍ ولا تحتاج إلى توصيف إضافي كما في حالة رمز التخصيص في التعريف السابق لأمر التخصيص في لغة الجافا. وللتفرقة بين الرموز النهائية والرموز غير النهائية فإن الاسم المجرد للرمز غير النهائي يتم وضعه بين الأقواس (< >) في شكل باكوس نور.

وبصفة عامة فإن التوصيف النحوي باستخدام شكل باكوس نور يتكون من مجموعة من القواعد (Rules) وكل قاعدة تقوم بوصف أو تعريف صيغة نحوية أو أمر محدد في لغة البرمجة ويمكن أن يكون هناك عدة تعريفات مختلفة لرمز غير نهائي واحد وذلك للتعبير عن صيغ نحوية مختلفة للرمز في لغة البرمجة والتعريف التالي يوضح ذلك عند توصيف أمر IF في إحدى لغات البرمجة:

< if-stmt > if < logic-expr > then < stmt >

< if-stmt > if < logic-expr > then < stmt > else < stmt >

ويمكن دمج أكثر من قاعدة نحوية في قاعدة نحوية واحده باستخدام رمز الاختيار (|) والتعريف التالي يقوم بدمج القاعدتين السابقتين في قاعدة نحوية واحده:

< if-stmt > if < logic-expr > then < stmt >

< if < logic-expr > then < stmt > else < stmt |

وبالرغم من بساطة شكل باكوس نور إلا أنه فعال وكافٍ لوصف جميع الصيغ النحوية الخاصة بلغات البرمجة المختلفة بما فيها الصيغ النحوية التي تحتوي على سلسلة أو قائمة (list) من التراكيب ولأي مستوى أو عمق مع إمكانية توضيح أولوية تنفيذ العمليات (Operators Precedence) وكيفية إتحاد العمليات (Operators Associativity) .

وصف القوائم (Lists)

إن القوائم مختلفة الطول (Variable Length Lists) تعتبر من النماذج التي يصعب وصفها ولكن شكل باكوس نور يستخدم كوسيلة مبتكرة لوصف القوائم التي تحتوي على عدد من العناصر النحوية في لغات البرمجة مثل الأمر الخاص بتعريف المتغيرات الذي يحتوي على عدة متغيرات غير محددة العدد ويتم تعريفها في أمر واحد داخل البرنامج كالمثال التالي:

;int Num1, Num2, Num3

فالأمر السابق يقوم بتعريف ثلاثة متغيرات (Num1,Num2,Num3) على أنها متغيرات صحيحة ولكن يمكن في نفس الأمر تعريف عدد من المتغيرات أكثر أو أقل من ثلاثة متغيرات و يستخدم شكل باكوس نور التكرار (Recursion) لوصف القوائم المتغيرة الطول كما في المثال التالي الذي يقوم بوصف قائمة من المتغيرات:

< var-list > < var >

< var > , < var-list > |

حيث أن الوصف السابق لقائمة المتغيرات (Variables list) يشمل على تكرار للرمز اللانهائي (<var-list>) والذي يتم تعريفه لكونه يظهر على يسار السهم ( ) كما يظهر أيضا على يمين السهم ( ) وهو ما يعني أن هذه القاعدة الوصفية هي قاعدة تستخدم التكرار فأي قاعدة يظهر فيها نفس الرمز اللانهائي على يسار و يمين السهم ( ) هي قاعدة تكرارية (Recursive Rule) كما في القاعدة السابقة والتي توضح أن قائمة المتغيرات إما أن تحتوي على متغير واحد فقط أو متغير يليه الرمز النهائي الفاصلة (,) ثم قائمة المتغيرات التي يمكن بدورها أن تتكون من متغير واحد أو من متغير يليه قائمة من المتغيرات وهكذا لأي عدد من المتغيرات فبهذا الوصف فإن قائمة المتغيرات يمكن أن تتكون من أي عدد من المتغيرات بدءً من متغير واحد أو أثنين أو ثلاثة أو أكثر.

القواعد النحوية والاشتقاق:

إن القواعد النحوية (Grammar) تعتبر وسيلة مولدة لتعريف اللغات فأي جملة تحتوي عليها اللغة يتم توليدها من خلال تطبيق القواعد النحوية الخاصة باللغة بشكل متتابع بدء من رمز خاص يطلق عليه رمز البدء (Start Symbol) حيث يطلق على هذه العملية عملية الاشتقاق (Derivation) وبصفة عامة فإنه للتأكد من أن أي جملة تنتمي للغة معينة فإنه لابد أن يتم اشتقاق هذه الجملة من القواعد النحوية الخاصة باللغة كما يمكن أيضا العكس عن طريق توليد أو اشتقاق جميع الجمل التي يمكن أن تنتمي إلى لغة معينة من القواعد النحوية الخاصة بهذه اللغة ولكنها تعتبر عملية صعبة جدا وخاصة عند زيادة عدد القواعد النحوية للغة كما في حالة اللغات الطبيعية فمن الصعب حصر جميع الجمل التي تنتمي إلى إحدى اللغات الطبيعية كاللغة العربية أو الإنجليزية وذلك لإتساع القواعد النحوية الخاصة بأي من هاتين اللغتين.

ولتوضيح عملية الاشتقاق نقدم المثال التالي الذي يتكون من مجموعة القواعد النحوية الخاصة بلغة برمجة بسيطة جدا تم وصفها باستخدام شكل باكوس نور

program> begin <stmt-list> end>

<stmt-list> <stmt>

<stmt> ; <stmt-list> |

<stmt> <var> = <expression>

var> A | B | C>

<expression> <var> + <var>

<var> - <var> |

<var > |

ومثل جميع لغات برمجة الحاسوب فإن رمز البداية (Start Symbol) الخاص بلغة البرمجة الموضحة في المثال السابق هو الرمز اللانهائي (<program>) حيث يستخدم هذا الرمز كرمز بداية لجميع لغات البرمجة وتبدأ عملية الاشتقاق من هذا الرمز.

ولغة المثال السابق تتكون من نوع واحد من الأوامر هو أمر التخصيص حيث أن أي برنامج مكتوب بهذه اللغة يبدأ بكلمة (begin) متبوعة بقائمة من الأوامر يفصل بينها رمز الفاصلة المنقوطة (;) وينتهي البرنامج بكلمة (end) . والتعبير الرياضي (Expression) في هذه اللغة إما أن يتكون من متغير واحد فقط أو من متغيرين يفصل بينهما رمز (+) أو رمز (-) واللذان يستخدمان في عمليتي الجمع والطرح على التوالي والاسم الخاص بمتغيرات هذه اللغة هي إما (A) أو (B) أو (C) ونوضح الآن كيفية اشتقاق برنامج من هذه اللغة البسيطة:

program > => begin < stmt-list > end >

begin < stmt > ; < stmt-list > end <=

begin < var > = < expression > ; < stmt-list > end <=

begin A = < expression > ; < stmt-list > end <=

begin A = < var > + < var > ; < stmt-list > end <=

begin A = B + < var > ; < stmt-list > end <=

begin A = B + C ; < stmt-list > end <=

begin A = B + C ; < stmt > end <=

begin A = B + C ; < var > = < expression > end <=

begin A = B + C ; B = < expression > end <=

begin A = B + C ; B = < var > end <=

begin A = B + C ; B = C end <=

ويلاحظ أن الاشتقاق يبدأ من رمز البداية الخاص باللغة (<program>) ورمز الاشتقاق (<=) يستخدم للانتقال من تركيبة إلى التركيبة التي تليها في عملية الاشتقاق وفي كل خطوة يتم استبدال أحد الرموز اللانهائية التي تحتوي عليها التركيبة بأحد التعريفات الخاصة به كما توضحه القواعد النحوية للغة وفي جميع الأحوال يتم في كل خطوة استبدال الرمز اللانهائي الذي يقع في أقصى يسار التركيبة للانتقال إلى الخطوة التالية وهو ما يسمى اشتقاق من أقصى اليسار (Left-Most Derivation) حيث يستمر الاشتقاق وحتى الوصول إلى تركيبة لا تحتوي على أي رموز غير نهائية وتحتوي بالكامل على مجموعة من الرموز النهائية فقط فعندها ينتهي الاشتقاق وتظهر الجملة أو البرنامج في حالة لغات البرمجة) التي تم اشتقاقها أو توليدها من القواعد النحوية الخاصة بهذه اللغة.

ويمكن أن يتم الاشتقاق من أقصى اليمين (Right-Most Derivation) أي باستبدال الرموز اللانهائية من اليمين إلى اليسار أو بأي ترتيب أخر دون أن يؤثر ترتيب الاشتقاق على اللغة التي يتم اشتقاقها من القواعد النحوية الخاصة بهذه اللغة ولكن الاشتقاق من أقصى اليسار هو أكثر أنواع الاشتقاق استخداما.

وفيما يلي مثال آخر يوضح القواعد النحوية الخاصة بجزء بسيط من إحدى لغات البرمجة :

< assign > < var > = < exp >

var > A | B | C >

< exp > < var > + < exp >

< var > * < exp > |

< exp > |

< var > |

وهذه القواعد تقوم بوصف أمر التخصيص الذي يحتوي الجانب الأيمن منه على تعبير رياضي يستخدم عملية الجمع والضرب وكذلك الأقواس وعلى سبيل المثال أمر التخصيص التالي:

(A= B * (A+C

يمكن توليده من القواعد السابقة عن طريق الاشتقاق من أقصى اليسار.

< assign > => < var > = < exp >

< A = < exp <=

< A = < var > * < exp <=

< A = B * < exp <=

(< A = B * (< exp <=

(< A = B * (< var > + < exp <=

(< A = B * (A + < exp <=

(< A = B * (A + < var <=

(A = B * (A + C <=

استخدام شجرة الإعراب (Using the Parse Tree)

من خصائص القواعد النحوية إمكانية وصف الهيكل النحوي لجمل أي لغة من اللغات في شكل هرمي وذلك باستخدام ما يسمى بشجرة الإعراب (Parse Tree) فعلى سبيل المثال فإن الشكل التالي يوضح الهيكل النحوي لأمر التخصيص الذي تم اشتقاقه في المثال السابق وذلك باستخدام شجرة الإعراب التالية:

ويلاحظ أن كل عقدة من العقد الداخلية (Internal Nodes) لشجرة الإعراب يتم عنونته بأحد الرموز اللانهائية بينما العقد الخارجية (External Nodes) أو ما يطلق عليها العقد الورقية (Leaf Nodes) فإنها تعنون بأحد الرموز النهائية.

غموض القواعد النحوية (Ambiguity) :

اذا كانت القواعد النحوية لأي لغة من اللغات تمكن من إنشاء أكثر من شجرة إعراب واحدة لأي من الجمل المشتقة من هذه اللغة فإن هذه القواعد تعتبر قواعد غامضة (Ambiguous) ولتوضيح ذلك فالمثال التالي يقوم بعرض نفس القواعد النحوية الخاصة بأمر التخصيص الموضح في المثال السابق ولكن بعد إجراء بعض التعديلات البسيطة عليه:

< assign > < var > = < exp >

Var > A | B | C >

< exp > < exp > + < exp >

< exp > * < exp > |

(< exp >) |

< var > |

فهذه القواعد النحوية الخاصة بأمر التخصيص في شكلها الأخير تعتبر قواعد غامضة وذلك لأن الجملة التالية:

A = B + C * A

وهي الجملة التى يمكن اشتقاقها من القواعد النحوية السابقة يمكن أن يتم إنشاء شجرتي إعراب لها كالموضحين في الشكل التالي مما يعني أن القواعد النحوية التي تم اشتقاق أو توليد هذه الجملة منها هي قواعد نحوية غامضة.

وغموض القواعد النحوية لأي لغة من اللغات يعتبر مشكلة بالنسبة للمترجم الذي يعتمد بشكل كبير على بعض دلالات القواعد النحوية الخاصة باللغة في الترجمة من هذه اللغة إلى اللغات الأخرى وخاصة إذا ما عرفنا أن المترجم الخاص بأي لغة يستخدم شجرة الإعراب الخاصة بكل تركيبة أو جملة مشتقة من هذه اللغة في توليد الترجمة المقابلة لهذه التركيبة أو الجملة في اللغة الأخرى فإذا كانت هناك أكثر من شجرة إعراب واحدة لأي جملة أو تركيبة في اللغة فإن المترجم الخاص بهذه اللغة لا يستطيع القيام بالترجمة لعدم إمكانية تحديد ترجمة مقابلة وحيدة لهذه الجملة أو التركيبة وهي المشكلة التي سيتم مناقشتها بالتفصيل في الأجزاء التالية.
أولويات تنفيذ العمليات (Operators Precedence) :

كما ذكرنا سابقا فإن القواعد النحوية تستطيع عند وصف التراكيب المختلفة لأي لغة من اللغات تقديم بعض الإيضاحات أوالدلالات لشجرة الإعراب المقابلة لكل تركيبة وهو ما يظهر بوضوح عند تحديد أولويات تنفيذ العمليات الحسابية المختلفة من جمع وطرح وضرب وقسمة داخل أي تعبير رياضي لأن العملية الحسابية التي تظهر في شجرة الإعراب في مستوى أدنى تكون لها الأولوية في التنفيذ عن العمليات الحسابية التي تظهر في مستوى أعلى ففي شجرتي الإعراب السابقتين نجد أن عملية الجمع لها أولوية في التنفيذ بحيث تسبق عملية الضرب في شجرة الإعراب التي تظهر في الجانب الأيمن بينما في شجرة الإعراب التي تظهر في الجانب الأيسر تكون عملية الضرب لها أولوية في التنفيذ عن عملية الجمع وذلك لأن عملية الجمع تظهر في شجرة الإعراب اليمنى في مستوى أدنى من مستوى عملية الضرب بينما في شجرة الإعراب اليسرى تظهر عملية الضرب في مستوى أدنى من مستوى عملية الجمع ومن هذا نستطيع فهم مدى الغموض والتعارض في المعلومات المستنتجة من كلا الشجرتين على الرغم من أنهما يعكسان الهيكل النحوي لأمر تخصيص واحد ولكنه مشتق من قواعد نحوية غامضة.

والقواعد النحوية لأي لغة يمكن أن تعطي جميع العمليات الحسابية نفس الأولوية في التنفيذ وتكون الأولوية فقط في التنفيذ لترتيب ورود العملية الحسابية داخل التعبير الرياضي بحيث يتم تنفيذ العمليات بالترتيب من اليمين إلى اليسار أو من اليسار إلى اليمين وذلك حسب طريقة تعريف التعبير الرياضي ولكن يمكن أيضا للقواعد النحوية عند تعريف الهيكل الخاص بالتعبير الرياضي أن نقوم بتحديد الأولوية الخاصة بكل عملية حسابية بغض النظر عن ترتيب ورودها داخل التعبير الرياضي وذلك عن طريق

إعطاء اسم مجرد مختلف لأطراف كل عملية حسابية لها أولوية تنفيذ مختلفة وهو ما سيتضح في المثال التالي:

< assign > < var > = < exp >

var > A | B | C >

< exp > < exp > + < term >

< term > |

< term > < term > * < factor >

< factor > |

(< factor > (< exp >

< var > |

والاشتقاق الخاص لأمر التخصيص التالي: A = B + C * A

يكون كما يلي وفقا للقواعد النحوية الخاصة بتعريف أمر التخصيص بشكلها الأخير:

< assign > => < var > = < exp >

< A = < exp <=

< A = < exp > + < term <=

< A = < term > + < term <=

< A= < factor > + < term <=

< A = < var > + < term <=

< A = B + < term <=

< A = B + < term > * < factor <=

< A = B + < factor > * < factor <=

< A = B+ < var > * < factor <=

< A = B + C * < factor <=

< A = B + C * < var <=

A = B + C * A <=

وشجرة الإعراب الوحيدة المقابلة لأمر التخصيص السابق تكون كما يلي:

حيث يتضح مدى التماثل أو التقارب بين الاشتقاق الخاص بأمر التخصيص المبين في المثال السابق وبين شجرة الإعراب المقابلة لنفس المثال كما يظهر أولوية تنفيذ عملية الضرب عن عملية الجمع لأن عملية الضرب تظهر في مستوى أدنى في شجرة الإعراب عن عملية الجمع وذلك كنتيجة لتحديد أولويات تنفيذ العمليات الحسابية في القواعد النحوية الخاصة بتوصيف أمر التخصيص في شكلها الأخير.

وكملاحظة أخيرة فإن كل عملية اشتقاق ناتجة من قواعد نحوية غير غامضة يقابلها شجرة إعراب وحيدة ولكن العكس غير صحيح لأن شجرة إعراب واحدة يمكن أن تقابل عددا من عمليات الاشتقاق المختلفة لنفس الجملة أو التركيبة وذلك لوجود عدة طرق مختلفة لاشتقاق الجملة أو التركيبة حسب ترتيب عملية الاشتقاق من أقصى اليسار أم من أقصى اليمين أو بأي ترتيب آخر ولتوضيح ذلك فإن الاشتقاق التالي لنفس أمر التخصيص المعطى في المثال السابق ولكنه في هذه المرة اشتقاق من أقصى اليمين وليس من أقصى اليسار كما في المرة السابقة:

< assign > => < var > = < exp >

< var > = < exp > + < term > <=

< var > = < exp > + < term > * < factor > <=

< var > = < exp > + < term > * < var > <=

var > = < exp > + < term > * A > <=

var > = < exp > + < factor > * A > <=

var > = < exp > + < var > * A > <=

var > = < exp > + C * A > <=

var > = < term > + C * A > <=

var > = < factor > + C * A > <=

var > = < var > + C * A > <=

var > = B + C * A > <=

A = B + C * A <=

ولكن مع ذلك فإن هذا الاشتقاق الأخير يقابله نفس شجرة الإعراب السابقة.
اتحاد العمليات (Associativity of Operators)

إن قدرة القواعد النحوية على وصف الإتحاد الخاص بالعمليات في التعبيرات بشكل سليم يعتبر أيضاً من الخصائص الهامة لهذه القواعد وذلك لأن صحة وصف إتحاد العمليات تنعكس على شجرة الإعراب عند تمثيل تعبير يحتوي على عدد من العمليات المتجاورة التي لها نفس الأولوية في التنفيذ ففي هذه الحالة فإن طريقة إتحاد كل عملية مع أطرافها تحدد ترتيب تنفيذ هذه العمليات المتجاورة وأمر التخصيص التالي يوضح ذلك:

A = B + C + A

فوفقاً للقواعد النحوية لأمر التخصص بشكلها الأخير الذي يوضح أولوية تنفيذ العمليات فإن شجرة الإعراب التي تمثل أمر التخصيص في المثال السابق تكون كما يلي:

حيث يتضح من شجرة الإعراب السابقة ووفقاً لقاعدة أن العملية التي تظهر في مستوى أدنى في شجرة الإعراب يكون لها أولوية في التنفيذ عن العملية التي تظهر في مستوى أعلى لذلك فإنه في المثال السابق يتم تنفيذ عملية الجمع اليسرى في التعبير الرياضي قبل عملية الجمع اليمنى أي أن ترتيب التنفيذ يكون من اليسار إلى اليمين عند ورود عدة عمليات جمع متجاورة داخل التعبير الرياضي وهو المتعارف عليه بالنسبة لعمليات الجمع أو الطرح أو الضرب أو القسمة وهو ما يطلق عليه إتحاد من ناحية اليسار أو إتحاد يساري (Left Associative) وهو المستخدم مع الحاسوب وبالنسبة لعملية الجمع فإنه من المعروف أن نوع الإتحاد سواء كان إتحاد يساري أو إتحاد يميني (Right Associative) فإن نتيجة عملية الجمع لن تختلف وذلك لأن عملية الجمع من الناحية الرياضية عملية متعادلة إتحادياً (Associative Operator) وهو ما يعبر عنه كما يلي:

$$(A + B) + C = A + (B + C)$$

ونفس الشيء بالنسبة لعملية الضرب ولكن الأمر يختلف بالنسبة لعمليتي الطرح والقسمة لأنهما عمليات غير متعادلة إتحادياً.

ويتم التعبير عن نوع الإتحاد في القواعد النحوية بحسب موضع تكرار الرمز اللانهائي في القاعدة النحوية التكرارية والتي يظهر فيها نفس الرمز اللانهائي على جانبي التعريف فإذا ظهر الرمز اللانهائي المتكرر في بداية الجانب الأيمن من القاعدة النحوية فهذا يعني أن اتحاد العملية يساري مثل القاعدة النحوية التالية :

< exp> < exp > + < term >

< term > |

والتي يظهر فيها الرمز اللانهائي المتكرر < exp > في بداية الجانب الأيمن للقاعدة وبالتالي يقوم بوصف إتحاد عملية الجمع على أنه إتحاد يساري ولكن إذا ظهر الرمز اللانهائي المتكرر في نهاية الجانب الأيمن من القاعدة النحوية فهذا يعني أن إتحاد العملية يميني مثل القاعدة النحوية التالية:

< factor> " < exp > ** < factor >

I <exp>

والتي توضح أن الإتحاد الخاص بعملية رفع الأس (Exponentiation) هو إتحاد يميني أي أن عمليات الأس المتجاورة يتم تنفيذها من اليمين إلى اليسار وليس العكس كما في حالة عمليات الجمع والطرح والضرب والقسمة وذلك يتضح من القاعدة النحوية السابقة حيث يظهر الرمز اللانهائي المتكرر <factor> في نهاية الجانب الأيمن للقاعدة.

3.1. شكل باكوس نور الموسع

(Extended Backus Naur Form (EBNF

بسبب بعض الإضافات التي تمت على شكل باكوس نور، فإنه يظهر ما يسمى بشكل باكوس نور المعدل أو الموسع وهو ما يطلق عليه اختصارا (EBNF) وعموماً فإن هذه الإضافات لم تؤثر على القوة الوصفية لشكل باكوس نور والتي ظهرت لنا في الجزء السابق ولكن هذه الإضافات كانت فقط لتسهيل عملية كتابة القواعد النحوية بهذا الشكل الجديد وكذلك تسهيل عملية قراءتها واستيعابها . ويمكن حصر الإضافات الشائعة الإستخدام في ثلاثة إضافات أساسية:

- أول إضافة هي إستخدام الأقواس ([ ]) للدلالة على أن ما بداخلها هو جزء اختياري في الصيغة النحوية التي يتم وصفها فمثلاً لوصف أمر f i الشرطي تكون القاعدة النحوية كما يلي :

[<if-Stmt> if < logic□expr > then < stmt > [else < stmt >

القاعدة السابقة توضح أن الجزء الأخير من أمر f i هو جزء اختياري وهو ما كان يحتاج في شكل الـ BNF الأساسي إلى وصف ذلك في أكثر من قاعدة نحوية وليست قاعدة نحوية واحدة كالموضحة في المثال السابق.

- الإضافة الثانية هي استخدام الأقواس ({ }) لتوضيح التكرار بدلاً من اللجوء إلى القواعد النحوية التكرارية وذلك يمكن من وصف القوائم في قاعدة نحوية واحدة دون الحاجة إلى استخدام عدة قواعد نحوية للقيام بذلك فمثلاً لوصف قائمة المتغيرات (< var-list >) فإنه يكفي لذلك القاعدة النحوية التالية:

{ <var □ list > < var> { , < var >

وهو ما يعني أن قائمة المتغيرات عبارة عن متغير واحد وقد يتبعه أي عدد من المتغيرات يفصل بينها الفاصلة (,) وذلك لأن عدد مرات التكرار قد يكون صفراً أو أكثر.

الإضافة الثالثة كانت لوصف عدة بدائل للاختيار من بينها حيث يتم وضع مجموعة الاختيارات بين الأقواس ( ) والفصل بينها برمز الاختيار (I) والقاعدة النحوية التالية تظهر كيفية القيام بذلك:

< term > < term > (* | / | %) < factor >

حيث توضح إمكانية الاختيار بين ثلاثة بدائل (* أو / أو %) و ذلك يتطلب وجود قاعدة نحوية منفصلة لكل اختيار من الاختيارات المختلفة.

والآن أصبحت الأقواس بأشكالها المختلفة ([ ] و { } و) جزء من الأدوات الوصفية لشكل باكوس نور المعدل وليست رموزاً نهائية كما كانت في شكل باكوس نور وتكمن المشكلة عند الاحتياج لاستخدام هذه الأقواس في الصيغ النحوية الخاصة بأحد اللغات ولكن هذه المشكلة يمكن التغلب عليها عن طريق وضع هذه الأقواس بين علامات التنصيص (´- ´) عند الاحتياج لاستخدامها كرمز نهائي وللتميز بينها وبين استخدامها كأداة وصفية في شكل باكوس نور المعدل.

وبالإضافة للإضافات الثلاثة الشائعة الاستخدام لشكل باكوس نور المعدل والتي تم توضيحها في العرض السابق يوجد بعض الإضافات الأخرى غير شائعة الاستخدام. هذا وقد تم حصر جميع الإضافات وإعداد القائمة الموحدة لشكل باكوس نور الموسع في نهاية عام1996 م ولكن لا يستخدم جميع عناصر القائمة الموحدة.

. القواعد الخصائصية Attribute Grammar

تعتبر القواعد النحوية الخصائصية امتداداً للقواعد خالية السياق ولكن تتميز عنها في أنها تمكن من وصف تراكيب لغات البرمجة بشكل أفضل وخاصة أن بعض خصائص التراكيب البرمجية يصعب وصفها باستخدام شكل باكوس نور. ومن أمثلة تلك الخصائص القواعد الخاصة بالتوافق النوعي:

(Type Compatibility) للبيانات فمثلاً في لغة الجافا لا يسمح بتخزين قيمة كسرية داخل متغير خاص بالقيم الصحيحة على الرغم من أن العكس مسموح به . فقاعدة مثل هذه يكون من الصعب وصفها باستخدام شكل باكوس نور وأيضاً نجد أن وصف الشرط الخاص بضرورة تعريف المتغيرات قبل استخدامها داخل أي برنامج فإنه من المستحيل وصف هذا الشرط باستخدام شكل باكوس نور أو حتى شكل باكوس نور المعدل وبصفة عامة فإن هذه النوعية من الخصائص تنتمي لما يسمى بقواعد الدلالات الساكنة(Static Semantics) ويطلق عليها ذلك الاسم لارتباطها وإن كان بشكل غير مباشر بدلالات البرنامج ولكن بسبب إمكانية تحديدها قبل البدء في تنفيذ البرنامج لذلك سميت دلالات ساكنة والتي يصعب أو قد يستحيل وصفها باستخدام شكل باكوس نور لذلك فقد تم إنشاء القواعد النحوية الخصائصية في نهاية الستينات من القرن الماضي حتى نتمكن من وصف الصيغ النحوية الخاصة بلغات البرمجة المختلفة بما تتضمنه من دلالات ساكنة.

والقواعد النحوية الخصائصية هي قواعد نحوية معتادة مثل القواعد الخالية السياق ولكن بعد إضافة عدد من الخصائص إليها بعض هذه الخصائص يتعلق بوصف الدلالات الساكنة والبعض الآخر يستخدم لتحديد كيفية حساب القيم الخاصة بكل خاصية وسوف يتم توضيح تلك الخصائص في الأجزاء التالية .

1.2 كيفية تعريف القواعد الخصائصية

المثال البسيط التالي الذي يوضح كيفية استخدام القواعد الخصائصية في وصف الدلالات الساكنة. وهو عبارة عن وصف القاعدة الخاصة بإحدى اللغات التي تشترط أن ينتهي كل أجراء (Procedure) داخل البرنامج باسم يتطابق مع الاسم الخاص بالإجراء والذي يتم تحديده مع بداية تعريف هذا الإجراء حيث يستخدم الرمز اللانهائي < proc-name> للتعبير عن اسم الإجراء بينما الخاصية < proc-name >. string فإنها تستخدم للدلالة عن قيمة الحروف الخاصة بهذا الاسم وللتفرقة بين الرمز اللانهائي < proc-name > عند ذكره عدة مرات داخل القاعدة النحوية فإنه يتم ترقيم الرمز برقم مختلف في كل مرة يذكر فيها بالقاعدة النحوية مع الملاحظة أن هذا الترقيم ليس جزءاً من الصيغة النحوية نفسها وفي هذه الحالة فإن القاعدة النحوية الخصائصية تكون كما يلي:

[syntax rule : < proc-def > procedure < proc-name >[1

< proc-body >

[end < proc-name >[2

= = predicate : < proc-name >[1].string

proc-name >[2].string <

ففي المثال السابق نجد القاعدة النحوية الثانية (predicate rule) وهي التي تستخدم الخاصية string للدلالة عن قيمة اسم الإجراء والتي تعتبر أحد خصائص الدلالات الساكنة وذلك لاشتراط أن اسم الإجراء لابد أن يتطابق عند بداية تعريف الإجراء مع الاسم المذكور عند نهاية التعريف. ونقدم الآن مثال آخر أكبر وأكثر شمولا للقواعد النحوية الخصائصية حيث يتضح من هذا المثال مدى إمكانية

استخدام القواعد الخصائصية في تأكيد التوافق النوعي للبيانات وذلك من خلال تعريف أمر التخصيص البسيط التالي:

< assign > < var > = < exp >

< exp > <var> + < var >

< var > |

var > A | B | C >

فالتعريف السابق عبارة عن قاعدة نحوية مجرده لا تتضمن أي دلالات ساكنة حيث يظهر من التعريف أن الجانب الأيمن لأمر التخصيص إما أن يكون متغير بسيط أو أن يكون تعبير رياضي أما الجانب الأيسر فلا بد أن يكون متغير والمتغيرات في التعريف إما أن تكون A أو B أو C أما التعبير الرياضي فهو على شكل متغير يضاف إلى متغير آخر ونريد الآن إضافة بعض الخصائص للتعريف السابق لتوضيح أن المتغيرات يمكن أن تكون من أحد نوعين أما أن تكون متغيرات صحيحة (Integer Variables) تقبل القيم الصحيحة فقط أو تكون متغيرات حقيقية (Real Variables) أي تقبل القيم الكسرية و عندما يكون هناك متغيران في الجانب الأيمن لأمر التخصيص ليس من الضروري أن يكونا من نفس النوع وعند ذلك يكون نوع التعبير الرياضي حقيقي وهو ما يعني أن ناتج التعبير الرياضي يكون كسري أما إذا كان المتغيران من نفس النوع سواء صحيح أو حقيقي فإن التعبير الرياضي يكون أيضاً مثلهما إما صحيح أو حقيقي على التوالي وفي جميع الأحوال لا بد أن يكون نوع المتغير الذي على يسار أمر التخصيص من نفس نوع التعبير الرياضي أو المتغير الذي على يمين أمر التخصيص فلا بد أن يكون طرفي أمر التخصيص الأيمن و الأيسر من نفس النوع ليكون أمر التخصيص سليماً لذلك سوف نستخدم القواعد النحوية الخصائصية للتعبير عن الدلالات الساكنة المشار إليها عن طريق إضافة بعض القواعد الدلالية (Semantic Rules) للتعريف السابق وستشمل تلك القواعد المضافة خاصيتان يرتبطان بالرموز اللانهائية.

الخاصية الأولى تتعلق بالنوع الفعلي (Actual Type) وهي خاصية ترتبط بالرمز اللانهائي لكل من المتغير(< var >) والتعبير الرياضي (< exp >) وتستخدم لتخزين النوع الفعلي لكلا منهما والتي إما أن تكون صحيحة(int) أو تكون حقيقية (real) في هذا المثال والتي يتم تحديدها لكل من المتغير أو التعبير الرياضي من خلال نوع الأطراف المتفرعة منهما في شجرة الإعراب.

أما الخاصية الثانية فتتعلق بالنوع المتوقع (Expected (Type للتعبير الرياضي (< exp >) وهي خاصية موروثة (Inherited Attribute) يتم تحديدها وفقاً لنوع المتغير الذي يظهر على يسار أمر التخصيص والذي إما أن يكون صحيح (int) أو حقيقي (real).

وفيما يلي نقدم النص الكامل للقواعد النحوية الخاصة بأمر التخصيص بعد إضافة القواعد الدلالية التي تعبر عن الدلالات الساكنة السابق شرحها:

<syntax rule : < assign > < var> = < exp -1

semantic rule : < exp>. expected-type <var>.actual-type

[syntax rule : < exp> <var> [2] + <var>[3 -2

semantic rule : < exp> . actual ⍰ type ←

(if (< var> [2] . actual-type = int

(and (< var> [3]. actual – type = int

then int

else real

predicate : <exp>.actual-type = = <exp>.expected-type

<syntax rule : <exp> <var -3

semantic rule : <exp>.actual-type <var>.actual-type

predicate: <exp>.actual-type = = < exp>.expected-type

syntax rule : <var> A l B l C -4

(semantic rule: <var>.actual-type look-up(<var>. string

ونلاحظ أن الوظيفة look-up تقوم بالبحث داخل جدول الرموز symbol table)) الخاص بالبرنامج عن المتغير المحدد باسمه والعودة بنوع هذا المتغير وشجرة الإعراب التالية تمثل أمر التخصيص (A = A + B) المشتق من القواعد النحوية الخصائصية السابقة :

والشكل التالي يوضح نفس شجرة الإعراب السابقة ولكن بعد إضافة الخصائص المستخدمة في القواعد الدلالية والمسار الخاص بكيفية الإستدلال عن القيمة الخاصة بكل خاصية والموضوع على الشكل باستخدام الخطوط المتقطعة (Dashed Lines)

وفيما يلي الخطوات الخاصة بتحديد قيمة كل خاصية من الخصائص المستخدمة في القواعد الخصائصية السابقة مرتبة وفقاً لترتيب تنفيذها:

(var>.actual-type look-up(A> -1 (من القاعدة 4)

exp>.expected-type <var>.actual-type> -2 (من القاعدة 1)

(var>[2].actual-type look-up(A> -3 (من القاعدة 4)

(var>[3]. actual-type look-up(B> -4 (من القاعدة 4)

exp>.actual-type int or real> -4 (من القاعدة2)

exp>.actual-type = = <exp>.expected-type> -5 (من القاعدة2)

حيث يلاحظ أن ترتيب تنفيذ الخطوات يسير داخل شجرة الإعراب من أسفل إلى أعلى في اتجاه المسار الموضح بالخطوط المتقطعة وأن القيمة الخاصة بالخطوة الأخيرة إما أن تتحقق في حالة تساوي النوع الفعلي للتعبير الرياضي مع النوع المتوقع له وإما أن لا يتحقق في حالة عدم حدوث هذا التساوي ونفرض أن نوع المتغير A هو متغير حقيقي (real) ونوع المتغير B هو متغير صحيح (int) فإن شجرة الإعراب بشكلها الأخير سيكون كما يلي بعد إجراء الخطوات السابقة وتحديد القيمة الخاصة بكل خاصية من الخصائص الموضحة على شجرة الإعراب السابقة:

حيث يتضح من الشكل تحقق التساوي بين كل من النوع المتوقع والنوع الفعلي للتعبير الرياضي مما يعني تحقق الشرط الخاص بالتوافق النوعي لأمر التخصيص والموضح بالقواعد الدلالية التي تعكس الدلالات الساكنة لأمر التخصيص في المثال السابق.

وعموماً التحقق من القواعد الدلالية الساكنة في أي لغة يعتبر من الأجزاء الأساسية لجميع المترجمات فإذا كان الشخص القائم بكتابة المترجم ليس لديه صعوبة في التعامل مع القواعد النحوية الخصائصية فإنه يستطيع استخدام أفكارها الأساسية لتصميم الجزء الخاص بهذا التحقق من الدلالات الساكنة للغة ولكن من أهم صعوبات استخدام القواعد الخصائصية في وصف القواعد النحوية والدلالية لأي لغة برمجة حقيقية هو حجمها وتعقيدها وخاصة أن زيادة عدد الخصائص والقواعد الدلالية المصاحبة لها يجعل هذه القواعد صعبة في كتابتها وكذلك قراءتها بالإضافة إلى صعوبة تتبع القيم الخاصة بهذه الخصائص داخل شجرة الإعراب التي تمثل هذه القواعد كل هذا قد يجعل من يقوم بتصميم وكتابة المترجم يفضل اللجوء إلى استخدام قواعد خصائصية أقل تعقيداً مع الاستعانة بأدوات مساعدة سهلة في كتابة المترجم عن إستخدام قواعد خصائصية معقدة لا تمكنه من استخدام مثل هذه الأدوات المساعدة.

3. وصف دلالات البرامج

في هذا الجزء نتحول إلى المهمة الصعبة الخاصة بوصف الدلالات المتغيرة (Dynamic Semantics) والتي يتم فيها تحديد معنى التعبيرات والأوامر والوحدات البرمجية المختلفة وهي المهمة التي يعتبر وصف الصيغ النحوية بالنسبة لها مهمة سهلة وذلك لتوافر الوسائل والطرق الخاصة بوصف الصيغ والقواعد النحوية وهي الشيء غير المتوفر بالنسبة لوصف الدلالات المتحركة فلا يوجد طريقة واحدة مقبولة ومتفق عليها للقيام بذلك وقد تم تسمية هذا النوع من الدلالات بالدلالات المتحركة لأن مدلولها لا يتم تحديده إلا عند تنفيذ البرنامج وليس قبل ذلك وأيضاً للتفرقة بينها وبين الدلالات الساكنة (Static Semantics) التي تم شرحها في الجزء السابق والتي يتم تحديد مدلولها أثناء عملية الترجمة وقبل البدء في تنفيذ البرنامج وبصفة عامة فإنه غالباً ما يطلق على الدلالات المتحركة لفظ دلالات فقط دون تحديد أنها متحركة وذلك لأن الأصل في الدلالات أنها متحركة ويتم تحديدها أثناء تنفيذ البرامج.

وهناك عدة أسباب للاهتمام بدلالات البرامج أول هذه الأسباب أن المبرمجين يحتاجون لمعرفة الوظيفة الخاصة بكل أمر أو وحدة برمجية حتى يتمكنون من كتابة البرامج وغالباً ما يستعينون في ذلك على قراءة الشرح المدون في الدليل الخاص باللغة والذي يكون مكتوباً باللغات الطبيعية سواء الإنجليزية أو الفرنسية أو حتى العربية ولكن هذا الشرح قد يكون غير كامل وغير محدد. أما السبب الآخر للاهتمام بدلالات البرامج وهو السبب الذي يعنينا الآن هو أن القائمين بكتابة المترجمات يحتاجون أيضاً إلى فهم الدلالات الخاصة بكل جزء من أجزاء اللغة الذي يقومون بإعداد المترجم الخاص بها ولصعوبة وتعقيد الوصف الرسمي لهذه الدلالات يكون اللجوء للوصف غير الرسمي باللغات الطبيعية ولهذا توجد العديد من الأبحاث في هذا الاتجاه تهدف إلى توفير أسلوب واضح يمكن الاعتماد عليه في وصف مثل هذه الدلالات البرمجية بشكل رسمي وذلك لتسهيل مهمة القائمين بكتابة المترجمات وأيضاً المبرمجين.

وبصفة عامة يوجد ثلاثة طرق أساسية لوصف دلالات البرامج حيث تقوم الطريقة الأولى والتي تسمى بطريقة الدلالات التشغيلية (Operational Semantics) بوصف معاني تراكيب اللغة عن طريق تأثير كل منها على حاسوب افتراضي. أما الطريقة الثانية والمسماة بطريقة الدلالات البديهية (Axiomatic Semantics) فهي طريقة تقوم باستخدام المنطق الرياضي والتي يعد وسيلة لإثبات مدى صحة البرامج والطريقة الثالثة والتي تسمى طريقة الدلالات الرمزية (Denotational Semantics) فهي تعتمد على تمثيل معاني تراكيب اللغة باستخدام الرموز الرياضية حيث يتم هذا التحويل لعناصر اللغة إلى مجموعة من الرموز الرياضية عن طريق عدد من الوظائف التكرارية (Recursive Functions) وسوف نكتفي في هذا الجزء بعرض الطريقة الأولى فقط بشيء من التفصيل.

1.3. الدلالات التشغيلية

الفكرة الأساسية لهذه الطريقة كما أشرنا سابقاً هي القيام بوصف معنى البرنامج عن طريق القيام بتنفيذ خطوات أو أوامر البرنامج على جهاز حاسوب سواء حقيقي أو افتراضي حيث يتم تحديد المعنى أو المدلول الخاص بكل أمر من أوامر البرنامج من خلال التغيرات التي تحدث على حالة جهاز الحاسوب عند تنفيذ هذا الأمر وتتمثل حالة جهاز الحاسوب في العدادات والمواقع التخزينية في ذاكرة الجهاز فإذا تم تسجيل حالة جهاز الحاسوب قبل تنفيذ الأمر ثم يتم تنفيذ الأمر المراد معرفة دلالاته وتسجيل حالة الجهاز بعد إتمام التنفيذ فعند ذلك تكون دلالة الأمر واضحة لأنها تتضح في مقدار التغير بين الحالة الأولى والثانية بجهاز الحاسوب والتي حدثت نتيجة لتنفيذ الأمر وتكمن المشكلة في هذه الطريقة بالرغم من سهولتها في إمكانية حدوث اختلاف في تأثير ومدلول الأمر حسب مواصفات الجهاز التي يتم اختبار مدى التغير الذي يحدث عند تنفيذ الأمر عليه بمعنى أن مدلول الأمر قد يختلف من جهاز حاسوب إلى آخر إذا اختلفت مواصفات الجهاز وهذه المشكلة يمكن التغلب عليها إذا تم استبدال جهاز الحاسوب الحقيقي بجهاز آخر افتراضي حيث يعتبر هذا الإجراء ليس بغريب ومستحسن في كثير من

الأحيان حيث يستخدم في تتبع خطوات البرامج لفهمها أو حتى للتأكد من صحتها ولتوضيح ذلك نقدم المثال التالي الذي يقوم بوصف مدلول أمر For في لغة السي باستخدام عدد من الأوامر البسيطة:

الدلالات التشغيلية للأمر أمر For في لغة السي

```
For (expr1 ; expr2 ; expr3 } (
        expr1 ;
        loop: if expr2 = = 0
        goto out


        expr3 ;
        {goto loop
        out   :
```

وعموماً فإن أول وأهم استخدام لهذه الطريقة هي وصف دلالات لغة الـ PL / 1 وذلك في بداية السبعينات من القرن الماضي حيث تم الاستعانة ببعض أوامر لغات البرمجة البسيطة والأقل في المستوى والتي ظهرت قبل ذلك لوصف مدلولات كل أمر وكل تركيبة في لغة الـ PL / 1 وهذه الطريقة بالرغم من سهولتها وبساطة أسلوبها إلا أن ما يعاب عليها هو الاعتمادية لأن وصف دلالات أي لغة باستخدام هذه الطريقة يعتمد على أوامر لغة أخرى معروف دلالاتها مسبقاً وليس على شكل موحد متفق عليه وهو ما لجأت إليه الطريقتان الأخرتان في الاعتماد على المنطق والرموز الرياضية في وصف دلالات التراكيب البرمجية المختلفة.

المترجم البسيط ذو المرحلة الواحدة

A Simple One Pass Compiler

1. الترجمة نحوية التوجه

إن القواعد الخالية السياق (Context ? Free Grammars) وشكل باكوس نور (Backus Naur Forum) اللذان تم عرضهما في الوحدة السابقة بجانب قدرتها على وصف القواعد النحوية للغات يمكن أيضاً استخدامها للمساعدة في توجيه عملية الترجمة الخاصة ببرامج هذه اللغات حيث تقود القواعد النحوية عملية الترجمة وهو الأسلوب الذي يسمى بالترجمة نحوية التوجه (Syntax ? Directed Translation) والشكل التالي يوضح مراحل هذا الأسلوب من الترجمة:

حيث يقوم محلل المفردات (Lexical Analyzer) بتحويل سيل الحروف الذي تدخل إليها إلى سيل آخر من الكلمات والرموز التي تمثل المفردات الخاصة باللغة التي يتم ترجمتها وهذا السيل من المفردات يتم إدخاله إلى المترجم النحوي التوجه (Syntax □ Directed Translator) الذي يقوم بتحويلها إلى شفرة وسيطة. ومن ذلك يتضح أن المترجم النحوي التوجه قد قام بدمج ثلاثة مراحل من مراحل الترجمة في وضعها المعتاد (مرحلة التحليل النحوي ومرحلة التحليل الدلالي ومرحلة توليد الشفرة الوسيطة) في مرحلة واحدة مجمعة تنتهي إلى الشفرة الوسيطة مباشرة.

ولتوضيح هذه الطريقة في الترجمة سوف نقوم ببناء مترجم بسيط لترجمة التعبيرات الرياضية بشكلها المعتاد الوسطيّ الشكل (Infix form) التي تظهر فيه العمليات الرياضية من جمع وطرح وضرب وقسمة في وسط أطراف العملية (مثل التعبير الرياضي 4 + 3-8) إلى شكل آخر من التعبيرات الرياضية يعرف بالشكل اللاحق (Postfix Form) والتي تظهر فيه العمليات الرياضية بعد أطرافها في التعبير الرياضي والشكل التالي يعرض نفس التعبير الرياضي السابق ولكن تظهر فيه العمليات في شكل لاحق: + 4 84

والسبب الأساسي للجوء إلى ترجمة لغة هذه التعبيرات البسيطة التي تشمل فقط على أرقام وعمليات رياضية هو لجعل مرحلة تحليل المفردات (Lexical Analysis) في غاية السهولة لأن كل حرف منفصل من سيل الحروف الذي سيتم إدخاله إلى هذه المرحلة يعتبر مفرده (Token) في حد ذاته من مفردات اللغة وذلك بفرض أن جميع أطراف عمليات التعبير الرياضي عبارة عن أعداد تتكون من رقم واحد فقط ولكن لاحقاً سوف نقوم بتوسيع هذه اللغة البسيطة لتشمل أعداداً تتكون من أرقام ومتغيرات وكذلك كلمات والآن نقوم بوصف التعبير الرياضي البسيط الذي يتكون من الأرقام بالإضافة إلى رمزي الجمع والطرح فقط وذلك باستخدام شكل ألـ BNF :

<exp> <exp> + <digit>

<exp> - <digit> |

<digit> |

digit> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9>

والقواعد النحوية السابقة توصف التعبير الرياضي بشكله الوسطيّ المعتاد والذي نريد تحويله باستخدام المترجم البسيط إلى تعبير رياضي بالشكل اللاحق والذي يمكن الآن تعريف كيفية التحويل من الشكل الأول إلى هذا الشكل الجديد:

أولاً: إذا كان التعبير الرياضي عبارة عن رقم أو حتى متغير فإن الشكل اللاحق له هو نفسه دون تغير.

ثانياً : إذا كان التعبير الرياضي يتكون من تعبيرين رياضيين بينهما عملية أي على الشكل E1 op E2 ،فإن الشكل اللاحق لهذا التعبير يكون على الشكل Ē 1 Ē 2op حيث أن Ē 1 ،Ē 2 هما الشكل اللاحق لكل من E1 ،E2 على التوالي.

ثالثاً : إذا كان التعبير الرياضي داخل أقواس أي على الشكل (E) فإن الشكل اللاحق له يكون Ē ، حيث أن Ē هو الشكل اللاحق للتعبير E دون استخدام الأقواس.

وسبب هذه النقطة الأخيرة هو أن التعبير الرياضي بشكله اللاحق لا يحتاج إلى استخدام الأقواس لتحديد أولوية تنفيذ العمليات لأن موضع العملية داخل التعبير الرياضي في هذا الشكل هو فقط الذي يحدد ترتيب تنفيذ العمليات ودون تطبيق قاعدة للأولويات أو الحاجة لاستخدام الأقواس وهو ما يميز هذا الشكل اللاحق للتعبيرات الرياضية فمثلاً الشكل اللاحق للتعبير الرياضي الوسطيّ 3+(4 - 8)+ 3 هو 3+ 84 بينما الشكل اللاحق للتعبير (3 + 4)-8 هو -+843

والآن نحتاج للقيام بهذه الترجمة باستخدام القواعد النحوية الخصائصية (Attribute Grammar) السابق شرحها في الوحدة السابقة وذلك لإضافة بعض الخصائص إلى القواعد النحوية التي تصف التعبير الرياضي المعتاد بشكله الوسطيّ والتي يتم تحديد القيمة الخاصة بكل منها عن طريق مجموعة من القواعد الدلالية يتم إلحاقها بالقواعد النحوية السابقة لتصبح كما يلي:

< Syntax rule: < exp > < exp >[1] + < digit -1

'+' Semantic rule: < exp >.post < exp >[1].post I I < digit >.post I I

< Syntax rule: < exp > < exp >[1] - < digit -2

'-' Semantic rule: < exp >.post < exp >[1].post I I <digit>. post I I

< Syntax rule: < exp > < digit -3

Semantic rule: < exp >.post < digit >.post

Syntax rule: <digit> 0 -4

'Semantic rule: < digit >.post '0

Syntax rule: <digit> 1 -5

'Semantic rule: <digit>.post '1

. . .

. . .

. . .

Syntax rule: <digit> 9 -13

'Semantic rule: <digit>.post '9

حيث يلاحظ أن الخاصية post تعبر عن القيمة الخاصة بها عن الشكل اللاحق للتعبير الرياضي بعناصره المختلفة وأن القواعد الدلالية التي تقوم بتحديد قيمة هذه الخاصية توصف كيفية ترجمة التعبير الرياضي من شكله الوسطي المعتاد إلى الشكل اللاحق هذا وقد تم استخدام الرمز || في القواعد الدلالية للتعبير عن عملية دمج عدة مفردات داخل الجملة أو التعبير الرياضي. مع ملاحظة أن القواعد الدلالية من رقم 4 إلى رقم 13 توضح أن الشكل اللاحق للأرقام من رقم 0 إلى رقم 9 هو نفس الرقم دون تغير.

والشكل التالي يعرض شجرة الإعراب ذات الحواشي (Annotated Parse-tree) للتعبير الرياضي 3 + 4-8 والتي تتضمن الحواشي الخاصة بها الخاصية post والتي تعكس قيمتها كيفية الوصول إلى الشكل اللاحق لهذا التعبير الرياضي.

وعموماً لا تقوم القواعد النحوية الخصائصية بتحديد ترتيب محدد لإيجاد القيم الخاصة بالخصائص التي تستخدم في قواعدها الدلالية. ففي المثال السابق لم يتم تحديد ترتيب محدد للوصول إلى الشكل اللاحق (قيمة الخاصية post) لكل عنصر من عناصر التعبير الرياضي فليس هناك فرقاً بين البدء من رقم 8 أو من رقم 4 أو من رقم 3 ولكن لا بد من إيجاد القيم الخاصة بخصائص جميع العقد المتفرعة من عقدة رئيسية قبل إيجاد القيم الخاصة بهذه العقدة الرئيسية فمثلاً في الشكل السابق لابد من تحديد قيمة الخاصية Post لكل من الرقم 8 والرقم 4 حتى يمكن الوصول لقيمة هذه الخاصية للتعبير 4 - 8 وهكذا حتى نصل إلى قيمة الخاصية الخاصة بالتعبير الرئيسي ولكن على الرغم من عدم تحديد هذا الترتيب فانه من المفضل أن يحدد مسبقا الترتيب الذي سيتم به إيجاد القيم الخاصة بالخصائص المستخدمة في القواعد الدلالية الملحقة بالقواعد النحوية الخصائصية وهو ما يعكسه ترتيب التنقل (Traversal) داخل شجرة الإعراب ذات الحواشي حيث يوجد أكثر من ترتيب يمكن التنقل من خلاله داخل أي شجرة بدء من جذر الشجرة (Root Node) ومروراً بجميع الأفرع التي تتضمنها الشجرة ولكن بالنسبة للتنقل داخل شجرة الإعراب بصفة خاصة بأن الترتيب المفضل لذلك هو إعطاء أولوية للوصول إلى عمق الشجرة وهو الأسلوب الذي يسمى التنقل للعمق أولا (Depth-First Traversal) حيث يتم التنقل داخل الشجرة بدءً من الجذر مع زيادة أبناء كل عقدة بشكل متكرر من اليسار إلى اليمين وهو ما يوضحه ترتيب الأرقام بالشجرة المبينة في الشكل التالي:

ووفقاً لهذا الأسلوب يكون ترتيب تنفيذ القواعد الدلالية المصاحبة للقواعد النحوية الخصائصية كما يوضحه خطوات الإجراء التكراري (Recursive Procedure) التالي:

(procedure visit (n: node

begin

for each child m of n, from left to right do

(visit (m

evaluate semantic rules at node n

end

.1. مخططات الترجمة

مخطط الترجمة (Translation Scheme) هو أيضاً عبارة عن قواعد خالية السياق ولكن مع إضافة بعض الخطوات البرمجية التي لها تأثيرات دلالية (Semantic Actions) حيث يتم وضع هذه الخطوات في الجانب الأيمن من القواعد النحوية وبصفة عامة تتشابه مخططات الترجمة مع القواعد الخصائصية في عديد من الأشياء ولكن هناك فرق أساسي بينهما هو أن ترتيب تنفيذ القواعد الدلالية في مخططات الترجمة يتم تحديدها بشكل واضح وذلك حسب موضع التأثير الدلالي داخل القاعدة النحوية حيث يتم وضع هذه التأثيرات بين أقواس من النوع ({ }) في الجانب الأيمن للقواعد النحوية وحسب مكان كل تأثير داخل القاعدة يتم تحديد ترتيب تنفيذ هذا التأثير بين التأثيرات الدلالية المختلفة والمثال التالي عبارة عن أحد القواعد النحوية التي تحتوي على تأثير دلالي:

[rest> + <term> {print ('+'} <rest> [1>

وتكمن وظيفة مخطط الترجمة في توليد مخرجات لكل جملة يمكن اشتقاقها من القواعد النحوية الخاصة باللغة التي يمثلها هذا المخطط وذلك عن طريق تنفيذ ما يحتوي عليه من تأثيرات دلالية بالترتيب الذي يعكس أسلوب التنقل للعمق داخل شجرة الإعراب الخاصة بهذه الجملة فعلى سبيل المثال إذا كانت شجرة الإعراب التالية تمثل القاعدة النحوية السابقة.

ففي هذه الحالة فإن التأثير الدلالي {print('+')} يتم تنفيذه بعد الانتهاء من جميع الأفرع المتشعبة من الشجرة التي تبدأ عند الرمز اللانهائي <term> ولكن قبل البدء في أفرع الشجرة التي تبدأ عند الرمز اللانهائي [1]<rest> وعموماً فإنه عند إنشاء شجرة إعراب لأحد مخططات الترجمة فإنه يتم تمثيل كل تأثير دلالي يحتوي عليه من خلال فرع إضافي يتم توصيله بخط متقطع مع الرمز اللانهائي الذي يقع في الجانب الأيسر من القاعدة النحوية التي تحتوي على هذا التأثير الدلالي وهو ما يظهر في شجرة الإعراب السابقة مع ملاحظة أن العقدة الخاصة بالتأثير الدلالي تكون عقدة منتهية مثلها في ذلك مثل العقد التي تمثل رموز نهائية في اللغة مثل رمز الجمع (+) في شجرة الإعراب السابقة.

والآن نعود إلى عملية ترجمة التعبيرات الرياضية من شكلها الوسطيّ المعتاد إلى الشكل اللاحق لها. ولكي يمكن استخدام مخططات الترجمة في القيام بهذه المهمة فإنه لا بد أن تكون القواعد النحوية الخصائصية التي تصف هذه العملية قواعد خصائصية بسيطة (Simple Attribute Grammar) ولكي تكون كذلك فلا بد أن تكون الترجمة الخاصة بكل رمز لا نهائي يظهر في الجانب الأيسر من كل قاعدة نحوية ينتج عن دمج الترجمات الخاصة بالرموز اللانهائية التي تظهر في الجانب الأيمن من تلك القاعدة النحوية وبنفس ترتيب ورودها في هذه القاعدة مع إمكانية إضافة أي رموز نهائية بين هذه الترجمات التي يتم دمجها فإذا كانت القواعد الخصائصية تتوفر فيها هذه الصفة فإنه يمكن اللجوء إلى مخططات الترجمة للقيام بعملية الترجمة المطلوبة وإن لم تكن فإنه لا يمكن اللجوء إلى مخططات

الترجمة للقيام بذلك ولتوضيح هذه الصفة نقوم باستعراض أحد القواعد الخصائصية الخاصة بالتعبير الرياضي والتي سبق عرضها في بداية هذا الجزء:

<Syntax rule : <exp> <exp>[1] + <digit

'+' || Semantic rule: <exp>.post <exp>[1].post | | <digit>.post

حيث نلاحظ أن ترتيب عملية ترجمة الرمز اللانهائي <exp> الذي يظهر في الجانب الأيسر للقاعدة النحوية السابقة هو ناتج عملية دمج لترجمات الرموز اللانهائية التي تظهر في الجانب الأيمن من هذه القاعدة وبنفس ترتيب ورودها. وهذا ما توضحه القاعدة الدلالية المصاحبة لهذه القاعدة مع إضافة الرمز النهائي '+' في نهاية هذه الترجمات وبالتالي يتضح أن القاعدة الخصائصية السابقة هي قاعدة خصائصية بسيطة ويمكن إعداد مخطط ترجمة مقابل لها يكون كما يلي :

{('+') exp> <exp> [1] + <term> { print>

حيث تم إضافة تأثير دلالي إلى القاعدة النحوية السابقة.

وبنفس الطريقة يمكن التأكد من أن جميع القواعد النحوية الخصائصية التي توصف عملية ترجمة التعبير الرياضي هي قواعد خصائصية بسيطة لذلك فإنه يمكن استخدام مخططات الترجمة للقيام بعملية ترجمة التعبيرات الرياضية من شكلها الوسطي المعتاد إلى الشكل اللاحق لها وتصبح مخططات الترجمة الخاصة بذلك كما يلي:

{ ('+') exp> <exp> + <digit> { print>

{ ('-') exp> <exp> - <digit> { print>

<exp> <digit>

{ ('digit> 0 { print ('0>

{ ('digit> 1 { print ('1>

. . .

{ ('digit> 9 { print ('9>

وبذلك تكون شجرة الإعراب التي تمثل التعبير الرياضي 3 + 4 -8 والتي تحتوي على تأثيرات دلالية كما يلي:

حيث يمكن الحصول على الترجمة الخاصة بهذا التعبير الرياضي والتي تمثل الشكل اللاحق له من خلال التنقل داخل شجرة الإعراب السابقة بأسلوب التنقل للعمق أولاً مع تنفيذ التأثيرات الدلالية وفقاً للترتيب الخاص بهذا الأسلوب لتكون نتيجة الترجمة التعبير الرياضي +3 84 والذي يمثل الشكل اللاحق لهذا التعبير الرياضي.

وبمقارنة شجرة الإعراب في حالة استخدام مخططات الترجمة بشجرة الإعراب في حالة استخدام القواعد النحوية الخصائصية نجد أن عملية الترجمة تتم في الحالة الأولى من خلال تنفيذ التأثيرات الدلالية والتي تقوم بطباعة مخرجات عملية الترجمة وفقا للترتيب المحدد لذلك بينما في الحالة الثانية تكون مخرجات عملية الترجمة ملحقة بجذر شجرة الإعراب ولا يمكن الوصول لها إلا بعد الانتهاء من إنشاء شجرة الإعراب بالكامل وإيجاد قيمة الخصائص الملحقة بها بشكل تتابعي من أسفل إلى أعلى لحين الوصول إلى الخاصية الخاصة بجذر الشجرة والتي تمثل نتيجة عملية الترجمة ولكن لكون ترتيب إنشاء شجرة الإعراب في الحالة الأولى والخاصة بمخططات الترجمة هو نفس ترتيب تنفيذ التأثيرات الدلالية الملحقة بشجرة الإعراب في هذه الحالة فإنه يمكن تنفيذ هذه التأثيرات أثناء إنشاء شجرة الإعراب ودون انتظار الانتهاء من بناء شجرة الإعراب بالكامل وهذا ما يميز طريقة مخططات الترجمة عن القواعد النحوية الخصائصية ولكن كما أشرنا لذلك من قبل فإنه حتى يمكن استبدال القواعد النحوية الخصائصية بمخططات الترجمة لا بد أن تكون تلك القواعد قواعد بسيطة.

. الإعراب Parsing

إن الإعراب كما أشرنا إلى ذلك من قبل هو العملية الخاصة بتحديد ما إذا كانت الجملة التي تتكون من بعض مفردات أحد اللغات يمكن توليدها أو اشتقاقها من القواعد النحوية الخاصة بهذه اللغة وفي هذا الصدد لا بد أن نلفت الانتباه إلى أن شجرة الإعراب التي تساعد في هذه العملية لا يقوم المترجم الخاص باللغة بإنشائها بشكل فعلي ولكن يتم محاكاة التنقل داخل هذه الشجرة دون الحاجة إلى إعدادها مسبقاً وهذا ما سوف يتضح في الجزء التالي حيث سيتم عرض برنامج كامل مكتوب بلغة السي يقوم بتنفيذ مخططات الترجمة المشار إليها في الجزء السابق بينما في هذا الجزء سوف نقوم بتقديم أحد طرق الإعراب التي يمكن تطبيقها عند بناء المترجمات نحوية التوجه.

أغلب الطرق الخاصة بالإعراب يمكن تصنيفها إلى مجموعتين طرق إعراب من أعلى إلى أسفل (Top-down Parsing) وطرق إعراب من أسفل إلى أعلى (Button-up Parsing) . ويرجع هذا التصنيف إلى ترتيب إنشاء عقد شجرة الإعراب. ففي المجموعة الأولى من طرق الأعراب يتم إنشاء هذه العقد بدءً من جذر الشجرة وتستمر حتى الوصول إلى الأوراق بينما في المجموعة الثانية يتم إنشاء عقد الشجرة بالعكس بدءً من أوراق الشجرة وتستمر في الصعود حتى الوصول إلى جذر الشجرة. وبصفة عامة فإن طرق الإعراب من أعلى إلى أسفل تعتبر هي الطرق الأكثر شيوعاً حيث يرجع ذلك إلى سهولة إنشاء شجرة الإعراب بالترتيب الطبيعي من أعلى إلى أسفل ومع ذلك فإن طرق الإعراب من أسفل إلى أعلى بالرغم من ذلك تستخدم في عدد ليس بالقليل من الأدوات البرمجية المساعدة التي تقوم بإنشاء المترجم أوتوماتيكيا من مخططات الترجمة وذلك لقدرتها على التعامل مع أغلب أنواع القواعد النحوية ومخططات الترجمة الخاصة بلغات البرمجة المختلفة.

1.1. الإعراب من أعلى إلى أسفل

في البداية سوف نقدم هذه الطريقة من خلال استخدام أحد القواعد النحوية المناسبة لهذا الأسلوب من الإعراب على أن نقوم فيما بعد في نهاية هذا الجزء بإعداد وحدة إعراب (parser) عامة تستطيع تطبيق هذه الطريقة على جميع أشكال القواعد النحوية فالقاعدة النحوية التالية تقوم بوصف بعض أنواع البيانات في لغة الباسكال (Language (Pascal :

<type> <simple>

<array [<simple>] of <type |

simple> integer>

char |

num dotdot num |

مع ملاحظة أن الرمز النهائي num يعنى أي رقم صحيح وتم اعتباره رمزاً نهائياً لتبسيط القاعدة النحوية الموضحة في المثال بينما الرمز النهائي dotdot فهو بديل للرمز '. .' لتوضيح ضرورة ورودهما كوحدة واحدة ولإنشاء شجرة الإعراب من أعلى إلى أسفل يتم البدء من جذر الشجرة المعنون برمز البداية اللانهائي مع الاستمرار في تطبيق الخطوة التالية:

عند كل موضع في الشجرة معنون بأحد الرموز اللانهائية يتم اختيار أحد القواعد النحوية التي يظهر هذا الرمز على الجانب الأيسر لها حيث يتم تكوين العقد المتفرعة من هذا الموضع والمعنونة بالرموز النهائية واللانهائية التي تظهر على الجانب الأيمن لهذه القاعدة التي تم اختيارها وهكذا إلى أن نصل إلى أوراق الشجرة المعنونة كلها برموز نهائية فقط.

والشكل التالي يوضح كيفية تطبيق الخطوة السابقة من البداية وحتى استكمال شجرة الإعراب الخاصة بهذه الجملة:

وتطبيق الأسلوب السابق لإنشاء شجرة الإعراب من أعلى إلى أسفل يمكن تنفيذه لعدد من القواعد النحوية أثناء مسح مجموعة الحروف المدخلة من اليسار إلى اليمين (في حالة اللغات الأجنبية) لمرة واحدة فقط وغالباً ما يطلق على مؤشر المفردة الحالية التي يتم مسحها اسم المؤشر المتقدم (lookahead) وفي بداية إنشاء شجرة الإعراب يكون موضع هذا المؤشر عند أول مفردة في جملة الإدخال من أقصى اليسار.

بالنسبة لمثال جملة الإدخال السابقة فإن المؤشر المتقدم يبدأ من عند كلمة 'array' بينما يتم البدء في إنشاء شجرة الإعراب المقابلة من جذر الشجرة المعنون برمز البداية اللانهائي <type> والشكل التالي يوضح موضع المؤشر المتقدم عند بعض مراحل إنشاء شجرة الإعراب الخاصة بالمثال السابق:

ونهدف إلى استكمال شجرة الإعراب بالطريقة التي يحدث فيها تطابق بينها وبين مفردات جملة المدخلات ولكي يحدث هذا التطابق تم في البداية إنشاء تفريعات من موضع جذر الشجرة تبدأ بالرمز النهائي array المقابل للمفردة التي يقف عندها مؤشر جملة الإدخال حيث توجد قاعدة واحدة من القواعد النحوية بالمثال التي يظهر فيها الرمز array في بداية الجانب الأيمن من القاعدة بينما يظهر الرمز اللانهائي < type> في الجانب الأيسر لها. وبالتالي يتم اختيار هذه القاعدة لإنشاء تفريعات شجرة الإعراب وهكذا يستمر إنشاء شجرة الإعراب. وفي كل مرة يكون فيها موضع الشجرة عند رمز نهائي يتطابق مع المفردة التي يشير لها مؤشر جملة المدخلات يتم تقديم هذا المؤشر إلى المفردة التالية بجملة الإدخال مع الانتقال إلى العقدة التالية في شجرة الإعراب. أما في حالة ما يكون موضع الشجرة عند رمز لانهائي، يتم اختيار القاعدة النحوية التي تحافظ على التطابق مع المفردة التي يشير لها مؤشر جملة الإدخال وهكذا إلى أن ينتهي إنشاء شجرة الإعراب التي تتطابق مع مفردات جملة المدخلات.

وبصفة عامة فإن عملية اختيار القاعدة النحوية المناسبة للرمز اللانهائي التي تحافظ على التطابق مع جملة المدخلات تحتمل التجربة والخطأ في بعض الأحيان وقد تتطلب العودة في الاختيار والبحث عن قاعدة نحوية أخرى إذا اتضح أن القاعدة التي تم اختيارها في المرة الأولي غير مناسبة ولكن هناك طريقة خاصة ومهمة للإعراب تسمى الإعراب التنبؤي (Predictive Parsing) لا يحدث فيها هذه المشكلة ولا نحتاج في هذه الطريقة التراجع في اختيار القاعدة النحوية لأن الاختبار يكون سليماً بصفة دائمة من المرة الأولى.

2.2. الإعراب التنبؤي

هناك طريقة من طرق الإعراب من أعلى إلى أسفل تدعى الإعراب التنازلي التكراري (Recursive-Descent Parsing) يتم فيها تحليل الصيغ النحوية الخاصة باللغة وإعداد مجموعة من الإجراءات التكرارية (Recursive Procedures) ليتم تنفيذها وفقاً لمفردات جملة المدخلات حيث يتم ربط كل إجراء من هذه الإجراءات بأحد الرموز اللانهائية التي تحتوي عليها القواعد النحوية للغة ويعتبر الإعراب التنبؤي أحد حالات هذه الطريقة حيث يتم اختيار الإجراء المصاحب لكل رمز لا نهائي دون أي غموض وفقاً للمفردة التي يشير لها مؤشر جملة المدخلات وبذلك يكون ترتيب تنفيذ هذه الإجراءات الخاصة بالتعامل مع جملة المدخلات تحاكي إنشاء شجرة الإعراب المقابلة لهذه الجملة ولكن دون أن يتم إنشاءها بشكل فعلي.

وفيما يلي نعرض مجموعة من الإجراءات تمثل عملية الإعراب التنبؤي وفقاً للقواعد النحوية المستخدمة في المثال السابق:

```
(procedure match (t:token *

begin
```

```
if lookahead = t then
lookahead : = nexttoken
else error
end
procedure type *
begin
if lookahead in {integer', ' char '. 'num'} then
simple
else if lookahead = 'array' then begin
;match ('array'); match ('['); simple
match ( ']'); match ('of'); type
end
else error
end
procedure simple *
begin
if lookahead = 'integer' then
('match ('integer
else if lookahead = 'char' then
('match ('char
else if lookahead = 'num' then
```

;('match ('num'); match ('dotdot'); match('num

else error

end

حيث تحتوي وحدة الإعراب التنبؤي السابقة على ثلاثة إجراءات: الإجراء الأول خاص بتنفيذ عملية التطابق مع مفردات جملة المدخلات بينما الإجراءين الثاني والثالث فهما يقابلان الرمزين اللانهائيين <type> و <simple> الموجودين بالقواعد النحوية السابقة وتبدأ عملية الإعراب باستدعاء الإجراء الخاص برمز البداية اللانهائي <type> مع نفس جملة المدخلات السابق استخدامها في المثال والتي يبدأ عندها المؤشر عند المفردة الأولى 'array' بهذه الجملة حيث يتحقق الأمر الشرطي الثاني في الإجراء type ليتم تنفيذ الخطوات التالية:

;match ('array'); match ('['); simple

match (']'); match ('of'); type

والتي تقابل الجانب الأيمن من القاعدة النحوية التالية :

<type> array [<simple>] of <type>

حيث يلاحظ أن كل رمز نهائي موجود بالجانب الأيمن للقاعدة السابقة يتطابق مع موضع مؤشر جملة المدخلات أما بالنسبة للرموز اللانهائية فإنه يتم استدعاء الإجراء الخاص بكل منها وعند استدعاء الإجراء الخاص بالرمز اللانهائي <simple> يتحقق الأمر الشرطي الثالث ليتم تنفيذ الخطوات التالية:

;('match ('num'); match ('dotdot'); match('num

والتي تقابل الجانب الأيمن من القاعدة النحوية التالية :

simple> num dotdot num>

حيث يقود مؤشر جملة المدخلات عملية الاختيار وهذا ما يحدث بسهولة عندما يبدأ الجانب الأيمن للقاعدة النحوية برمز نهائي أي بأحد مفردات اللغة لأنه في هذه الحالة يحدث التطابق مباشرة مع موضع مؤشر جملة المدخلات إما في حالة بدء الجانب الأيمن للقاعدة النحوية برمز لا نهائي وهو ما يحدث عند تحقيق القاعدة التالية:

<type> <simple>

وهي القاعدة التي يتم استخدامها عندما يكون موضع مؤشر جملة المدخلات يقف عند المفردة 'integer' وعندها يتم استدعاء الإجراء type ولعدم وجود قاعدة نحوية يظهر فيها الرمز اللانهائي <type> في الجانب الأيسر من القاعدة والرمز النهائي 'integer' في بداية الجانب الأيمن منها لذلك فإن القاعدة السابقة تستخدم في هذه الحالة وهو الذي يتم بالفعل حيث يقوم الإجراء type باستدعاء الإجراء simple عندما يكون موضع مؤشر جملة المدخلات يقف عند المفردة 'integer' . وفي هذه الحالة فإن طريقة الإعراب التنبؤي تعتمد على معلومات تساعدها في القيام بذلك من خلال معرفة جميع الرموز النهائية التي يمكن أن تبدأ بها جميع مشتقات الرمز اللانهائي الذي يظهر في بداية القاعدة النحوية حيث تم تحديد أن الرموز النهائية 'integer' و 'char' و 'mum' يمكن أن تشتق من الرمز اللانهائي <simple> ولكي نكون أكثر تحديداً فإذا فرض أن الرمز اللانهائي الذي يظهر في بداية الجانب الأيمن من القاعدة النحوية هو الرمز α فإن (α)FIRSTه هي المجموعة الأولي للرمز α التي تشتمل على جميع الرموز النهائية التي تبدأ بها جميع الجمل التي يمكن اشتقاقها من الرمز اللانهائي α بمعنى أن

{FIRST (<simple>) = {integer, char, num

{FIRST (array [<simple>] of <type>) = {array

مع ملاحظة أنه يمكن اشتقاق الجملة الفارغة والتي يرمز لها بالرمز ϵ من بعض الرموز اللانهائية وبالتالي فإن (FIRST (α قد تشتمل على هذا الرمز وعموماً فإننا سندرس المجموعة الأولي وطريقة حسابها في الوحدة الخامسة ولكننا نشير هنا إلى حالة وجود أكثر من قاعدة نحوية تشترك في الرمز اللانهائي الذي يظهر في الجانب الأيسر لهما كما في حالة القاعدتين التاليتين :

A> α>

A> β>

ففي هذه الحالة ولكي ينجح الإعراب التنبؤي فإنه لا بد من عدم حدوث تقاطع بين محتويات (α)FIRSTه ، ومحتويات (β)FIRSTه لأن حدوث ذلك يؤدي إلى عدم التأكد عند اختيار أي من القاعدتين التي يتم تطبيقها في حالة ما إذا كان موضع مؤشر جملة المدخلات يقف عند المفردة المشتركة بين الاثنين ولذلك فإن الإعراب التنبؤي لا يمكن استخدامه مع القواعد النحوية التي تحدث فيها مثل هذه الحالة.

. مشكلة التكرار من ناحية اليسار

إن وجود تكرار من ناحية اليسار (Left Recursion) في القواعد النحوية الخاصة باللغة من الممكن أن يسبب مشكلة لطريقة الإعراب التنازلي التكراري (بما فيها أسلوب الإعراب التنبؤي) وهذه المشكلة تتعلق بإمكانية حدوث تكرار غير منتهى عند تنفيذ الإجراءات الخاصة بوحدة الإعراب ولتوضيح ذلك نفترض القاعدة النحوية التالية :

<exp> <exp> + <term>

ففي هذه القاعدة نجد أن الرمز اللانهائي <exp> والذي يظهر في أقصى يسار الجانب الأيمن من القاعدة النحوية هو نفس الرمز اللانهائي الذي يظهر في الجانب الأيسر لها وبفرض أن الإجراء exp المقترن بهذا الرمز اللانهائي قرر استخدام هذه القاعدة النحوية لكون الجانب الأيمن لها يبدأ بنفس الرمز فإنه سيقوم باستدعاء نفسه مرة أخرى وهو ما يؤدي إلى حدوث التكرار إلى ما لا نهاية وذلك لأن مؤشر جملة المدخلات لا يتغير موضعه للأمام إلا في حالة تطابق موضع المؤشر مع الرمز النهائي الموجود بالقاعدة النحوية ولكن القاعدة السابقة تبدأ برمز لا نهائي وبالتالي لن يحدث أي تقدم في مؤشر جملة المدخلات مما يجعل التكرار غير منتهى.

وللتغلب على هذه المشكلة فإنه يمكن إعادة كتابة القاعدة التي تتضمن على تكرار من ناحية اليسار ليصبح التكرار من الناحية اليمين (Right Recursion) وذلك لأن هذا النوع الأخير من التكرار لا يسبب مشكلة لطريقة الإعراب التنازلي التكراري ولكنه قد يصعب من عملية ترجمة جمل المدخلات التي تحتوي على عمليات يسارية الإتحاد (Left-associative Operators) وهو ما سنعرضه في الأجزاء التالية ونوضح الآن كيفية تحويل القواعد النحوية التي تتضمن على تكرار من ناحية اليسار إلى قواعد نحوية ذات تكرار من ناحية اليمين وبفرض أن الرمز اللانهائي <A> يتم وصفه من خلال القاعدة التالية:

| A> <A> α β>

حيث أن كلا من α و β عبارة عن سلسلة من الرموز النهائية واللانهائية دون أن يبدأ أي منهما بالرمز اللانهائي <A> والقاعدة النحوية الأولى في هذه الحالة هي قاعدة ذات تكرار من ناحية اليسار ويمكن إعادة كتابة القاعدتين السابقتين ليصبحا كما يلي :

<A> β <R>

R> α <R> | Є>

فهذه القواعد النحوية الأخيرة تكافئ القاعدتين السابقتين لكونهم يحققوا نفس النتيجة ولكن مع استبدال التكرار من ناحية اليسار إلى تكرار من ناحية اليمين والذي يظهر في القاعدة الثانية من المجموعة الأخيرة حيث يظهر الرمز اللانهائي <R> والذي تم إضافته إلى هذه المجموعة في الجانب الأيسر لتلك القاعدة وأيضا في أقصى يمين الجانب الأيمن لها وهو ما يمثل تكرار من ناحية اليمين ونطبق الآن ذلك على المثال التالي الذي يشمل على قاعدتين نحويتين ذات تكرار من ناحية اليسار :

<exp> <exp> + <term>

<term> |

ففي هذا المثال نجد ما يلي :

<A> = <exp>

< α = + <term

< β = <term

و بالتالي يمكن استبدال القاعدتين السابقتين إلى مجموعة القواعد النحوية التالية ذات التكرار من ناحية اليمين:

<exp> <term> <rest>

<rest> + <term> <rest>

Є |

مع إمكانية إعادة كتابة القاعدة النحوية الثانية من أجل التبسيط لتصبح كما يلي :

rest> + <exp> | Є>

3. إنشاء مترجم للتعبيرات البسيطة

الآن وبعد دراسة الترجمة نحوية التوجه وطرق الإعراب من أعلى إلى أسفل وخاصة طريقة الإعراب التنازلي التكراري التي تتضمن أسلوب الإعراب التنبؤي فإنه يمكننا إنشاء مترجم نحوي التوجه والذي يقوم بترجمة التعبيرات الرياضية بشكلها الوسطي المعتاد إلى الشكل اللاحق لها حيث سيتكون هذا المترجم من برنامج كامل مكتوب بلغة C وكما ذكرنا في السابق فإننا سوف نكتفي في هذه المرحلة بالتعبيرات الرياضية التي تتكون من مجموعة من الأرقام التي يفصل بينها رمزي الجمع والطرح فقط على أن يتم تعقيد هذه التعبيرات في الأجزاء التالية من هذه الوحدة لتتضمن الإعداد والمتغيرات والعمليات الأخرى. وعموماً تساعد مخططات الترجمة نحوية التوجه في تحديد مواصفات المترجم وسوف نستخدم المخططات التالية والسابق عرضها من قبل كتعريف للمترجم الذي نحن بصدد إعداده :

{('+') exp> <exp> + <term> {print>

{('-')exp> <exp> - <term> { print>

<exp> <term>

{('term> 0 {print ('0>

{('term> 1   {print ('1>

. . .

{('term> 9   {print ('9>

وكما يحدث في أغلب الحالات من ضرورة إدخال بعض التعديلات على القواعد النحوية حتى يمكن إعرابها بواسطة الإعراب ألتنبؤي فانه بالنسبة للقواعد النحوية السابقة تحديداً لا بد من الاستعاضة عن التكرار من ناحية اليسار والذي لا يستطيع الإعراب ألتنبؤي التعامل معه ولكن لا بد أن يتم ذلك بحرص شديد ولتوضيح ذلك فإن القواعد النحوية السابقة ستصبح كما يلي بعد استبعاد التكرار من ناحية اليسار وباستخدام نفس الطريقة المشار إليها من قبل:

<exp> <term> <rest>

rest> + <exp> | - <exp> | Є>

term> 0 | 1 | . . . | 9>

وعلى الرغم من أن هذه القواعد السابقة بعد حذف التكرار من ناحية اليسار منها تقوم بتوليد نفس اللغة التي يمكن توليدها من هذه القواعد النحوية قبل حذف هذا التكرار إلا أن هذه القواعد النحوية في وضعها الأخير تعتبر غير مناسبة لترجمة التعبيرات الرياضية إلى الشكل اللاحق لها حيث تكمن مشكلة هذه القواعد النحوية في أن القاعدة الثانية منها لا تقوم بتحديد طرفي كل من عملية الجمع أو عملية الطرح وبالتالي فإن إضافة التأثيرات الدلالية إلى مثل هذه القواعد يكون من الصعب القيام به بالشكل السليم الذي يعكس عملية الترجمة ولهذا فإن طريقة حذف التكرار من ناحية اليسار لا بد من استكمالها حتى يمكن تطبيقها في حالة احتواء القواعد النحوية على بعض التأثيرات الدلالية فإذا كانت القاعدة النحوية التي تحتوي على تكرار من ناحية اليسار على الشكل:

A> <A> α | <A> β | γ>

فإنها ستصبح على الشكل التالي بعد حذف التكرار من ناحية اليسار :

<A> γ <R>

R> α <R> | β <R> | Є>

وفي حالة تطبيق هذه الطريقة بعد تعميمها على القواعد النحوية الخاصة بترجمة التعبيرات الرياضية والتي تحتوي على بعض التأثيرات الدلالية فإن المقابل لكل من α و β و γ سيكون كما يلي:

<A> = <exp>

{('+') α = + <term> {print

{('-') β = - <term> {print

< γ = <term

وبالتالي فإن مخطط الترجمة السابق سيصبح بعد حذف التكرار من ناحية اليسار كما يلي :

<exp> <term> <rest>

<rest> + <term> {print ('+')} <rest>

<term> {print ('-')} <rest> - |

Є |

{('term> 0 {print ('0>

{('term> 1 {print ('1>

. . .

{('term> 9 {print ('9>

وهذا المخطط الأخير للترجمة يجعل من الممكن استخدام طريقة الإعراب التنازلي التكراري حيث نقوم الآن بإعداد المترجم بلغة السي الذي يعتمد على هذا المخطط السابق حيث تتضمن شفرة البرنامج التالي الوظائف المقترنة بكل من <exp> و <rest> و <term> وهي الرموز اللانهائية الموجودة في تلك المخطط : -

exp

}

; term ; rest

{

rest

```
}
} ('+' = = if ( lookahead
;match ( '+' ); term
;  putchar ('+') ; rest
{
} ( '-' = =else if ( lookahead
;  match ( '-' ) ; term
;  putchar ('-' ) ; rest
{
; else
{
term
}
} ( (if ( isdigit (lookahead
;(putchar ( lookahead
;(match (lookahead
{
; else error
{
```

وبالنسبة للوظيفة match التي تقوم بالتأكد من تطابق المفردة التي نقوم باختبارها مع مؤشر جملة المدخلات ولأن اللغة التي نحن بصدها جميع مفرداتها تتكون من حرف واحد، لذلك فإن مهمة هذه الوظيفة هي مقارنة وقراءة الحروف فقط.

وقبل تقديم البرنامج الكامل المكتوب بلغة السي والذي يقوم بترجمة التعبيرات الرياضية البسيطة إلى الشكل اللاحق لها سوف ندخل بعض التحسينات على الوظائف الثلاثة السابقة من هذا البرنامج وخاصة الوظيفة rest حيث سيعاد كتابتها لتصبح كما يلي وذلك لتحويلها من وظيفة تكرارية (Recursive Function) إلى وظيفة غير تكرارية (Iterative Function) تحتوي على تكرار داخلي لتكون أسرع في التنفيذ.

```
rest
}
(while (1
} ('+' ==if ( lookahead
;  match ('+') ; term
; ( '+' ) putchar
{
} ('-' = = else if ( lookahead
;  match ( '-' ) ; term
; ( '-' ) putchar
{
; else break
{
```

والآن نقدم البرنامج الكامل الذي يقوم بعملية ترجمة التعبيرات الرياضية البسيطة التي تتكون من مجموعة من الأرقام يفصل بينها رمزي عملية الجمع وعملية الطرح فقط وذلك من شكلها المعتاد إلى

الشكل اللاحق لها وهذا البرنامج مكتوب بلغة السي ويتضمن الوظائف الثلاثة السابقة بالإضافة إلى الوظيفة الرئيسية main والتي يتم استدعائها في بداية تشغيل البرنامج وكذلك كل من الوظيفة match السابق الإشارة إليها والوظيفة error التي تقوم بطباعة رسالة الخطأ عند حدوث ذلك:

```
<include <ctype.h #

;int lookahead

main

}

;  lookahead = getchar

;  exp

; ('putchar ( '\n

{

exp

}

;  term  ; rest

{

rest

}

( while ( 1

} ( ' + ' = = if (lookahead

;('+') match ('+'); term  ; putchar

{

} ('-' = = else if (lookahead
```

```
; ('-' ) match ( '-') ; term  ; putchar
{
; else break
{
term
}
} ((if ( isdigit (lookahead
; (putchar(lookahead
; (match (lookahead
{
; else error
{
(match (int t
}
(if (lookahead == t
; lookahead = getchar
;  else error
{
error
}
; ( " printf (" syntax error \ n
```

; (exit (1

{

. تحليل المفردات Lexical Analysis

في هذا الجزء سوف نضيف إلى المترجم الذي تم تقديمه في الجزء السابق الوحدة المسئولة عن تحليل المفردات وهي الوحدة التي تقوم بتحويل جملة المدخلات التي تتكون من مجموعة من الحروف إلى مجموعة من مفردات اللغة التي تنتمي لها هذه الجملة وبالتالي فإن مخرجات هذه الوحدة هي مدخلات وحدة الإعراب (Parser) وتكمن أهمية محلل المفردات (Lexical Analyzer) في تبسيط القواعد النحوية التي تعتمد عليها وحدة الإعراب للقيام بالمهمة الخاصة بها والذي يتضح في كون الرموز النهائية في هذه القواعد عبارة عن مفردات (Tokens) وعدم وجود مثل هذه الوحدة داخل المترجم يصعب من مهمة وحدة الإعراب ويجعلها تتعامل مع الحروف وليس المفردات ولهذا السبب كان تعاملنا في الجزء السابق مع التعبيرات الرياضية التي تتكون فقط من مجموعة من الأرقام ورمزي الجمع والطرح والتي يعنى أن جميع مفرداتها تتكون من حرف واحد فقط ولكي نتمكن من ترجمة التعبيرات الرياضية التي تشمل على إعداد ومتغيرات فإن الأمر يستلزم وجود الوحدة الخاصة بتحليل المفردات ضمن هذا المترجم وسنبدأ هذا الجزء بمجموعة من الوظائف التي لا بد أن يتضمنها محلل المفردات.

1.4. حذف المسافات البيضاء والتعليقات

في بداية تحليل المفردات لا بد من استبعاد جميع الحروف التي لن يتم ترجمتها من جملة المدخلات وذلك للتركيز فقط على باقي الحروف التي تمثل أجزاء مفردات اللغة ومن أهم الحروف التي يتم استبعادها المسافات البيضاء (White Spaces) وهي جميع الحروف التي تشغل حيز فارغ داخل جملة المدخلات وتتضمن هذه الحروف المسافات الفردية (Blanks) والمسافة المتعددة (Tabs) وكذلك الحروف الخاصة بالانتقال إلى سطر جديد (New Lines) حيث يقوم محلل المفردات بحذف جميع المسافات البيضاء من جملة المدخلات التي سيتم ترجمتها وكذلك التعليقات (Comments) التي يتم إضافتها لجمل البرامج في العديد من لغات البرمجة وذلك لكونها لا يتم ترجمتها ولهذا يتم معاملتها مثل المسافات البيضاء.

2.4 القيم الثابتة

إن أطراف العمليات في أي تعبير رياضي يمكن أن تكون قيم ثابتة (Constants) والتي تمثل الأعداد الصحيحة أو الكسرية وبالنسبة للأعداد الصحيحة فإن كل عدد يتكون من مجموعة من الأرقام ومن الممكن السماح لمثل هذه القيم الثابتة الصحيحة أن تتواجد داخل التعبيرات الرياضية عن طريق إضافة بعض القواعد إلى القواعد النحوية التي تصف هذه التعبيرات أو بإنشاء مفردة خاصة تمثل هذه الثوابت الصحيحة داخل القواعد النحوية الخاصة بالتعبيرات الرياضية والأسلوب الأخير هو الأقرب للاستخدام في أغلب المترجمات حيث يقوم محلل المفردات بتحويل مجموعة الأرقام التي تكون القيمة الثابتة الصحيحة إلى وحدة واحدة تمثل المفردة الخاصة بمثل هذه الثوابت الصحيحة فإذا افترضنا أن الرمز النهائي num هو تلك المفردة التي تمثل الأعداد الصحيحة فعندما تظهر سلسلة من الأرقام المتتابعة داخل جملة المدخلات فإن محلل المفردات يقوم باستبدالها بالمفردة num لتصل إلى وحدة الإعراب في هذه الصورة الأخيرة مع ملاحظة أن القيمة الأصلية لهذه المفردة والخاصة بقيمة العدد

الصحيح فإنه لا بد من تمريرها أيضاً لوحدة الإعراب حيث تعتبر هذه القيمة هي إحدى خصائص هذه المفردة التي تسمى num .

3.4. تمييز المتغيرات والكلمات المفتاحية

إن كل لغات البرمجة تسمح باستخدام المتغيرات (Variables) في البرامج المكتوبة بهذه اللغات حيث يتم إعطاء اسم (Identifier) لكل متغير لكي يتم استخدامه داخل هذه البرامج والقواعد النحوية للغات البرمجة غالباً ما تقوم بمعالجة هذه المتغيرات وأسمائها كأحد مفردات تلك اللغات ووحدة الإعراب وفقاً لهذه القواعد تحتاج أن ترى نفس المفردة ولتكن id في كل مرة يظهر فيها اسم أحد المتغيرات داخل جملة المدخلات فعلى سبيل المثال إذا كانت المدخلات كالتالي:

; SUM = SUM + X

فإن هذه الجملة لابد أن يتم تحويلها بواسطة محلل المفردات لتصبح :

; id = id + id

والتي تستخدم داخل وحدة الإعراب ولكن الأمر يتطلب التفرقة بين المتغير SUM والمتغير X واللذان يظهران في الجملة الأصلية وهو ما لم يتضح في جملة المدخلات المستخدمة بواسطة وحدة الإعراب حيث تم استبدال جميع المتغيرات بدون تمييز بينها بالمفردات id وحل هذه المشكلة يتم بنفس الأسلوب التي تمت به معالجة قيم الثوابت الصحيحة حيث لا بد أن يصاحب المفردة id في كل مرة تظهر في جملة المدخلات الخاصة بوحدة الإعراب خاصية تقوم بتحديد اسم المتغير المرتبط بهذه المفردة.

أما بالنسبة للكلمات المفتاحية (Keywords) وهي الكلمات المحجوزة التي تستخدم في جميع لغات البرمجة مثل if و for و while وغيرها من الكلمات التي يكون لها مدلول خاص داخل اللغة فإن مثل هذه الكلمات لا بد لوحدة تحليل المفردات أن تفرق بينها وبين أسماء المتغيرات وليتم ذلك فإن كل لغات البرمجة لا تسمح بإعطاء أسماء للمتغيرات يشابه الكلمات المفتاحية المحجوزة للغة وبالتالي فإن محلل المفردات يعتبر سلسلة الحروف المتتابعة هو اسم لأحد المتغيرات فقط إذا لم تتطابق سلسلة هذه الحروف مع أحد الكلمات المفتاحية أما إذا حدث التطابق فإنه يعتبرها هي الكلمة المفتاحية التي تتكون من هذه الحروف المتتابعة لكن هناك أيضا مشكلة أخرى تتمثل في كيفية الفصل بين مفردات اللغة عند ظهور أكثر من مفردة بشكل متتابع في جملة المدخلات وخاصة في حالة اشتراك عدد من الكلمات المفتاحية في بعض الحروف التي تتكون منها وهي المشكلة التي سنعرض طريقة التعامل معها لا حقاً.

4.4. بناء محلل المفردات

عند إضافة محلل المفردات بين وحدة الإعراب وسلسلة المدخلات يكون التفاعل بينها بالأسلوب الموضح في الشكل التالي :

حيث يتضح من الشكل أن محلل المفردات يقوم بقراءة الحروف بشكل متتابع وتجميعها في مجموعة لتكون حروف مفردات اللغة ثم يقوم بإرسال كل مفردة على حدة مع الخصائص التي تعبر عن قيمتها إلى وحدة الإعراب ولكن في بعض الأحيان قد يضطر محلل المفردات إلى قراءة بعض الحروف الإضافية قبل أن يستطيع تحديد المفردة التي تمثلها هذه الحروف فعلى سبيل المثال عند قراءة الحرف '<' فإنه لا بد من قراءة الحرف التالي لتحديد ما إذا كانت تمثل المفردة الخاصة بعملية المقارنة ' أكبر من ' أم تمثل جزء من المفردة الخاصة بعملية المقارنة ' أكبر من أو تساوي ' فإذا كان الحرف التالي هو الحرف '=' فهذا يعني أنها تمثل المفردة الثانية أما إذا كان بداية حرف مفرده جديدة فإن ذلك يعني أنها تمثل المفردة الأولى والخاصة بعملية المقارنة ' أكبر من ' وفي هذه الحالة الأخيرة فإن محلل المفردات يكون قد قرأ حرف زيادة خاص بمفردة أخرى لذلك من الضروري إرجاع هذا الحرف الزائد إلى سلسلة المدخلات مرة أخرى حتى ينتظم أداء محلل المفردات ويستمر بشكل سليم.

أما بالنسبة للتفاعل بين محلل المفردات ووحدة الإعراب فإن محلل المفردات غالبا ما يقوم بإرسال المفردات التي يحققها إلى وحدة الإعراب مفردة تلو الأخرى وليست كمجموعة من المفردات مرة واحدة لذلك فإن المعتاد أن تقوم وحدة الإعراب باستدعاء محلل المفردات لطلب مفردة جديدة مع تحديد خصائصها المختلفة وسوف نقوم الآن ببناء محلل المفردات الذي سيسمح للتعبير الرياضي أن يتضمن على مسافات بيضاء مثل التي سبق شرحها وكذلك إمكانية استخدام الأعداد وليست الأرقام فقط كأطراف للعمليات الحسابية التي يتضمنها التعبير الرياضي على أن نؤجل التعامل مع المتغيرات إلى الجزء التالي.

والوظيفة التالية عبارة عن محلل للمفردات للتعبير الرياضي البسيط مكتوب بلغة السي يقوم بإهمال المسافات البيضاء التي تظهر في التعبير وكذلك تجميع الأعداد:

```
<include <stdio.h#
< include < ctype.h #
;int lineno =1
; int tokenval = NONE
()int lexan
}
;int t
} (while (1
; t=getchar
('if( t == ' ' | | t == '\t
```

```
;
('else if ( t == '\n
; lineno = lineno + 1
} ( (else if ( isdigit (t
; '0tokenval = t – '
; t = getchar
} ( (while ( isdigit(t
; 'tokenval = tokenval * 10 + t - '0
; t = getchar
{
; (ungetc (t , stdin
; return NUM
{
} else
; tokenval = NONE
; return t
{
{
{
```

حيث تقوم الوظيفة lexan بالتكرار وفي كل مرة يتم قراءة حرف. فإذا كان هذا الحرف مسافة فردية أو متعددة لا يتم إرجاع أي مفردة لوحدة الأعراب ويستمر في التكرار. أما إذا كان الحرف هو الحرف الخاص بالانتقال لسطر جديد فإنه يتم زيادة المتغير lineno لمتابعة رقم سطر المدخلات دون إرجاع

أي مفردة لوحدة الإعراب. أما في حالة قراءة سلسلة من الأرقام التي تكون أحد الأعداد فسيتم إرجاع المفردة num مع تحديد قيمة العدد ووضعها في المتغير tokenval . ما عدا ذلك من حروف فإنه يتم إرجاعها كما هي إلى وحدة الإعراب على اعتبار أنها تمثل إحدى العمليات الحسابية.

ويجب الأخذ في الاعتبار أنه للسماح باستخدام الأعداد مع التعبيرات الرياضية فإن الأمر يتطلب إجراء تغير في القواعد النحوية الأخيرة الخاصة بالتعبير الرياضي بحيث يتم وصف الرمز اللانهائي <term> بالشكل التالي:

(<term> (<exp>

{ (num { print(num.value |

بدلا من وصفه عن طريق الأرقام المنفصلة كما كان في السابق وبناءاً على ذلك فإن الإجراء الخاص بهذا الرمز اللانهائي <term> في وحدة الإعراب سوف يتغير ليصبح كما يلي :

```
term
}
} ( ' ) ' = = if ( lookahead
;( '(' )match( '(' ); exp; match
{
} ( else if (lookahead = = NUM
;(printf ("%d", tokenval); match(NUM
{
;() else error
{
```

. إدماج جدول الرموز

دائما ما يستخدم إحدى هياكل البيانات تسمى جدول الرموز (Symbol Table) وذلك للاحتفاظ بمعلومات عن تراكيب لغة برنامج المصدر المختلفة هذه المعلومات يتم تجميعها أثناء مراحل التحليل الخاصة بالمتر جم وذلك لاستخدامها خلال مرحلة التجميع لتوليد شفرة البرنامج المستهدف فمثلا أثناء تحليل المفردات يتم تخزين الحروف التي تكون اسم أحد المتغيرات داخل إحدى خانات جدول الرموز

على أن تستكمل باقي المعلومات الخاصة بهذا المتغير من نوع وطريقة ومدى الاستخدام ومكان تخزين القيمة الخاصة به خلال المراحل التالية من عملية الترجمة حيث نستخدم تلك المعلومات أثناء توليد الشفرة المستهدفة وذلك لتحديد الشفرة المناسبة لتخزين واستخدام هذا المتغير.

وسنقوم الآن في هذا الجزء باستعراض كيفية تفاعل محلل المفردات مع جدول الرموز وذلك عن طريق استخدام بعض الوظائف التي تقوم بتخزين أو استرجاع تلك الرموز من جدول الرموز فعند تخزين أحد الرموز يتم أيضا تخزين مفردة اللغة المرتبطة بهذا الرمز والعمليتان التاليتان يتم تنفيذهما على جدول الرموز :

هذه العملية تقوم بإضافة خانة جديدة داخل جدول الرموز insert (s , t)i يخزن فيها مجموعة الحروف s التي تكون الرمز وكذلك المفردة t التي ينتمي إليها على أن تعود برقم فهرس هذه الخانة.

هذه العملية تبحث عن الخانة التي تحتوي على الرمز الذي lookup (s)i يتكون من مجموعة من الحروف s وتعود برقم فهرس هذه الخانة إذا كان موجود وألا يكون رقم الفهرس 0 .

ويقوم محلل المفردات باستخدام العملية الثانية لتحديد ما إذا كان أحد الرموز قد ظهر من قبل ولديه خانة في جدول الرموز أم أنه يظهر لأول مرة وعند ذلك يستخدم العملية الأولى لإضافة خانة جديدة له في جدول الرموز.

1.5. معاملة الكلمات المفتاحية المحجوزة

إن الكلمات المفتاحية المحجوزة (Reserved Keywords) يتم التعامل معها بإضافة خانة خاصة لكل منها داخل جدول الرموز عند إنشاؤه وقبل ظهور أي رموز أخرى وبالتالي لا يمكن تسمية أي متغير باسم يشابه أحد هذه الكلمات فمثلا إذا كانت اللغة تتضمن الكلمات المحجوزة for و while كمفردات في هذه اللغة بنفس الأسماء for و while فعند بداية إنشاء جدول الرموز يتم تنفيذ العمليتين الآتيتين:

;(insert ("for",for

;(insert ("while", while

وعند ذلك فإن أي عملية lookup("for")i تالية تحدد أنها تنتمي لمفردة اللغة for وبالتالي لا يمكن تسمية أي متغير بهذا الاسم في هذه اللغة.

2.4. تطبيق جدول الرموز

الآن وبعد استخدام جدول الرموز يمكن استخدام المتغيرات داخل التعبير الرياضي وهو ما سيؤدي إلى إجراء بعض التعديلات على الوظيفة الخاصة بمحلل المفردات لتصبح كما يلي بعد إعادة كتابتها بلغة السي:

<include <stdio.h #

```
<include <ctype.h#
;[char lexbuf [BSIZE
;int lineno = 1
;int tokenval = NONE
int lexan
}
;int t
} (while (1
; t = getchar
( 'if ( t == ' ' | | t == '\t
;
('else if (t == '\n
;lineno = lineno + 1
} ( (else if ( isdigit(t
;(ungetc(t , stdin
;(scanf("%d", &tokenval
;return NUM
{
} ( (else if ( isalpha(t
; int p, b =0
} ( (while ( isalnum(t
```

```
;lexbuf[b] = t
;t = getchar
;b = b + 1
(if (b >= BSIZE
;error
{
;lexbuf[f] = EOS
(if (t ! = EOF
;(ungetc(t , stdin
;(p = lookup(lexbuf
(if (p == 0
;(p = insert(lexbuf, ID
;tokenval = p
;return symtable[p].token
{
(else if (t == EOF
;return DONE
} else
;tokenval = NONE
;return t
{
```

{

{

حيث يلاحظ أن محلل المفردات في نسخته الأخيرة عندما يقرأ حرف أبجدي يقوم بتخزين الحروف الأبجدية أو الرقمية داخل المخزن الوسيط lexbuf وعند الانتهاء من تجميع حروف الاسم يقوم بالبحث عنه داخل جدول الرموز باستخدام العملية lookup ولأن جدول الرموز يحتوي على جميع الكلمات المحجوزة للغة منذ إنشائه فإنه لن يسمح أن يكون هذا الاسم خاص بأحد المتغيرات وبالتالي يتعامل معه على أنه كلمة مفتاحية لها مدلول خاص باللغة إما إذا لم تكن كذلك ولم تظهر من قبل يتم إضافته إلى جدول الرموز باستخدام عملية insert على اعتبار انه اسم لأحد المتغيرات وعند ذلك يتم إرسال قيمة الفهرس الخاص بهذا المتغير داخل جدول الرموز إلى وحدة الإعراب عن طريق المتغير tokenval وكذلك ما يفيد أن هذه الحروف تمثل اسماً لمتغير.
تحليل المفردات

LEXICAL ANALYSIS
1. عملية المسح

إن وظيفة محلل المفردات هو قراءة حروف من برنامج المصدر ليشكل منها وحدات منطقية يتم التعامل معها في الأجزاء التالية من المترجم وهذه الوحدات المنطقية التي يكونها محلل المفردات هي مفردات (Tokens) اللغة المكتوب بها البرنامج وتشبه عملية تشكيل الحروف في مفردات عملية تشكيل الحروف على هيئة كلمات في إحدى جمل اللغة الإنجليزية والتي تستلزم اتخاذ القرار المناسب عند تشكيل كل كلمة ليكون معناها متسق مع سياق الجملة الموضوعة فيها.

ومفردات أي لغة هي عبارة عن عناصر أساسية يتم دائماً تعريفها كأحد أنواع البيانات التي يتم حصر القيم التي من الممكن أن تأخذها في عدد محدد من القيم وهي الأنواع التي تسمى أنواع معددة (Enumerated Types) فمثلاً يمكن تعريف المفردات بلغة السي كما يلي:

typedef enum

{ ,IF, FOR, WHILE, PLUS, MINUS, NUM, ID}

;tokentype

ومفردات لغة البرمجة يتم تصنيفها في عدة مجموعات تشمل مجموعة الكلمات المحجوزة مثل IF التي يتم تمثيلها بتدفق الحروف "if" وكذلك تشمل مجموعة الرموز الخاصة مثل الرموز الرياضية PLUS و MINUS اللذان يتم تمثيلهما بالحروف "+" و "-" . وأيضاً تشمل مجموعة المفردات التي يتم تمثيلها بسلاسل حروف متعددة مثل ID و NUM والتي تمثل المعرفات و الأعداد .

وعموماً فإن المفردات كوحدات منطقية يجب أن يتم التفرقة بينها وبين الحروف التي تمثلها. فمثلاً هناك فرق بين المفردة IF و بين الحروف "if" التي تمثل هذه المفردة ولكي تكون هذه التفرقة واضحة يطلق على الحروف التي تمثل المفردة قالب المفردة (Lexeme) وبعض المفردات يكون لها قالب واحد مثل

الكلمات المحجوزة بينما يوجد لبعض المفردات عدد غير محدد من القوالب التي يمكن تمثيل المفردة بها مثل الأرقام والمعرفات وفي هذه الحالة الأخيرة فإن تدفق الحروف التي تمثل المفردة لا بد من الاحتفاظ بها لذلك فإن المترجم يحتفظ بهذه الحروف فى جدول يسمى جدول الرموز (Symbol Table) أثناء عملية المسح التي تحدث خلال مرحلة تحليل المفردات وعموما فإن قيمة هذه الحروف التي تمثل كل مفردة تعتبر أحد الخصائص (Attributes) التي ترتبط بالمفردة حيث يوجد خصائص أخرى للمفردات يتم إضافتها لكل مفردة موجودة في جدول الرموز حيث تحدث هذه الإضافة خلال مراحل متفرقة من الترجمة.

وعلى الرغم من أن مهمة وحدة تحليل المفردات هي تحويل برنامج المصدر بالكامل إلى سيل من المفردات ولكن نادراً ما يتم ذلك كله مرة واحدة وإنما الذي يحدث غالباً وكما ذكرنا ذلك من قبل في الوحدة الثالثة فإن محلل المفردات وتحت تحكم وحدة الإعراب يقوم بإرجاع مفردة واحدة في كل مرة يتم فيها استدعائه من خلال الوظيفة التي يمكن أن تأخذ الشكل التالي في لغة السي:

; (tokentype getToken (void

فهذه الوظيفة المعرفة بالشكل السابق عندما يتم استدعائها تقوم بإرجاع المفردة التالية حسب المدخلات التي لديها كما تقوم بتحديد الخصائص الإضافية لهذه المفردة وغالبا لا تحصل هذه الوظيفة على سلسلة الحروف المدخلة في شكل معطيات (Parameters) ولكنها تحتفظ بهذه المدخلات في مخزن وسيط (Buffer) وكمثال لذلك فإن وظيفة getToken تعمل بافتراض وجود السطر التالي في شفرة برنامج المصدر:

a [index] = 4 + 2

وبفرض أن سطر المدخلات السابق تم تخزينه في مخزن وسيط للمدخلات كالتالي :

حيث يشير السهم إلى موضع حرف المدخلات التالي فعند استدعاء وظيفة getToken فإنها تحتاج الآن إلى إهمال المسافات الأربعة التي تظهر في بداية المدخلات ثم تقوم بالتعرف على حرف الـ "a" كقالب لمفردة يتكون من حرف واحد. وهذا القالب عبارة عن اسم لأحد المعرفات لذلك فإن الوظيفة ستقوم بإرجاع القيمة ID التي تعبر عن نوع هذه المفردة وتترك مخزن المدخلات الوسيط كالتالي :

ومن خلال الاستدعاء التالي لوظيفة getToken يتم تكرار نفس الخطوات للتعرف على فتحة القوس التي تأخذ الحرف "[" وهكذا إلى أن يتم تحليل مفردات سطر المدخلات بالكامل.
. التعبيرات المنتظمة

إن التعبيرات المنتظمة (Regular Expressions) هي إحدى طرق تمثيل القوالب الخاصة بسلاسل الحروف وكل تعبير منتظم r يتم تعريفه بالكامل بواسطة مجموعة سلاسل الحروف التي يتطابق معها. وهذه المجموعة تسمى اللغة التي يتم توليدها من هذا التعبير المنتظم ونرمز لها بالرمز L(r)i ولا يوجد علاقة محددة بين لغات التعبيرات المنتظمة ولغات البرمجة وذلك لأن الأولى لغات بسيطة ولا تستخدم إلا في تمثيل أجزاء من لغات البرمجة، كما سنتعرف على ذلك فى الأجزاء التالية. وعموماً فإن لغة التعبير المنتظم تعتمد إلى حد كبير على مجموعة الحروف والرموز التي تمثل أبجديات (Alphabet) اللغة والتي غالبا ما تشمل جزءاً أو حتى كل الحروف التي تتكون منها شفرة الـ ASCII وأبجديات أي لغة يرمز لها بالرمز اللاتيني ∑ (سيجما) وقد يكون لبعض حروف أبجديات أحد اللغات عدة معاني. وذلك على حسب موقعها داخل التعبير المنتظم. لهذا يجب التفرقة بين شكل الحرف ومعناه وأيضا قد يحتوي التعبير المنتظم على بعض الحروف التي يكون لها دلالات خاصة داخل التعبير حيث يطلق على هذه الحروف لفظ حروف أولية (Metacharacters) وهذه الحروف في الواقع لا يمكن أن تتضمن عليها أبجديات أي لغة أو على الأقل يجب التمييز بينها بطريقة أو بأخرى.

1.2. تعريف التعبيرات المنتظمة

سنقوم الآن بوصف مضمون التعبيرات المنتظمة من خلال استعراض اللغات التي يمكن توليدها من كل قالب من قوالب هذه التعبيرات. وسوف يتم ذلك على مراحل مختلفة حيث سنبدأ بوصف مجموعة من التعبيرات المنتظمة البسيطة التي تتكون من رموز منفصلة ثم يتم وصف العمليات التي تولد تعبيرات منتظمة جديدة من التعبيرات القائمة وذلك كما يحدث مع التعبيرات الرياضية. وسنكتفي في هذا الجزء بأقل مجموعة من التعبيرات المنتظمة التي تحتوي فقط على العمليات والحروف الأولية الأساسية ولكن فى الجزء القادم سنضيف مجموعة أخرى من التعبيرات المنتظمة.

1.1.2 التعبيرات المنتظمة البسيطة

هذا النوع من التعبيرات المنتظمة يتكون فقط من حروف مفردة من حروف أبجديات اللغة التي تتطابق مع نفسها. بفرض أن الحروف "a" من حروف الأبجدية ∑ فإن التعبير المنتظم a يتطابق مع الحرف "a" وهذا ما نعبر عنه بأن:

{L(a) = {a

بالإضافة إلى ذلك هناك رمزان يتم استخدامهما في بعض الحالات الخاصة. الرمز الأول يستخدم للتعبير عن السلسلة الفارغة من الحروف (Empty String) وهي السلسلة التي لا تحتوي على أية حروف حيث يرمز لها بالرمز ϵ (ايبسلون) ويتم التعبير عن ذلك بأن :

{L(ϵ) = {ϵ

والرمز الثاني هو حالة اللغة الفارغة التي لا تحتوي على أي سلاسل من الحروف حيث يستخدم الرمز Ø في ذلك ويتم تعريف هذه اللغة كما يلي :

{ } = (L(Ø

ويجب ملاحظة أن هناك فرقاً بين الحالتين السابقتين لأن الحالة الأولى تمثل اللغة التي تحتوي فقط على السلسلة الفارغة من الحروف بينما في الحالة الثانية فإن اللغة لا تحتوي على أي سلاسل من الحروف ولا حتى السلسلة الفارغة وعموماً فإن كلاهما يمثلان حالتان خاصتان من اللغات.

2.1.2 عمليات التعبيرات المنتظمة

يوجد ثلاثة عمليات أساسية للتعبيرات المنتظمة: العملية الأولى خاصة بالاختيار بين البدائل والتي يرمز لها بالحرف الأولي "|" والعملية الثانية خاصة بالدمج (Concatenation) وهي التي يعبر عنها بالتتالي كما سيتضح من الأمثلة أما العملية الثالثة فخاصة بالتكرار (Repetition) والتي يرمز لها بالحرف الأولى "*" وسنعطي الآن تراكيب المجموعة المقابلة لهذه العمليات وذلك عن طريق تقديم اللغات التي تتطابق معها.

أولاً : الاختيار بين البدائل

إذا كان كل من r و s تعبيران منتظمان فإن r | s هو التعبير المنتظم الذي يتطابق مع أي سلسلة من الحروف تتطابق أما مع التعبير المنتظم r أو التعبير المنتظم s وبخصوص اللغات التي يتم توليدها من ذلك فإن لغة التعبير المنتظم r | s تتكون من إتحاد لغتي التعبيران المنتظمان r و s وهو ما يتم التعبير عنه كما يلي:

(L(r | s) = L(r) U L(s

وعلى سبيل المثال بفرض التعبير المنتظم a | b فإنه يتطابق مع أحد الحرفين a أو b بمعنى أن:

{L(a | b) = L(a) U L(b) = {a} U {b} = {a, b

أما التعبير المنتظم a |Є فإنه يتطابق مع السلسلة التي تتكون من حرف واحد هو 'a' أو السلسلة الفارغة التي لا تحتوي على أي حروف بمعنى أن:

{ L(a | Є) = {a , Є

مع ملاحظة أن عملية الاختيار بين البدائل لا تقتصر فقط على الاختيار بين بديلين ولكن تمتد للاختيار بين أكثر من بديلين فمثلا:

{L(a | b | c | d) = { a, b, c, d

وفي حالة وجود متتابعة من البدائل فإن النقاط الثلاثية "" قد تستخدم في التعبير عن ذلك فمثلاً التعبير المنتظم a | b | | z يتطابق مع أي من الحروف الأبجدية الصغيرة بدءً من حرف 'a' وحتى حرف 'z' .

ثانياً : الدمج :

أن عملية الدمج بين التعبيران المنتظمان r و s والذي تكتب بالتعبير المنتظم rs والذي يتطابق مع جميع سلاسل الحروف الناتجة عن دمج سلسلتين من الحروف الأولى تتطابق مع التعبير المنتظم r والثانية تتطابق مع التعبير المنتظم s فمثلاً التعبير المنتظم ab يتطابق فقط مع سلسلة الحروف "ab" بينما التعبير المنتظم (a | b) c فإنه يتطابق مع سلستي الحروف "ac" و "bc" وهو ما يمكن التعبير عنه كما يلي:

{L((a | b) c) = L(a | b) L(c) = {a, b} {c} = {ac,bc

ويمكن ان تتضمن عملية الدمج أكثر من تعبيرين منتظمين بمعنى أن :

(L(rn (rn) = L(r1) L(r2 L(r1 r2

ثالثاً : التكرار

عملية تكرار التعبير المنتظم والتى تكتب r*r وذلك لإجراء التكرار على التعبير المنتظم r حيث يتطابق التعبير المنتظم r*r مع أي عدد منتهي من عمليات الدمج المتتالية لسلاسل الحروف التي تتطابق مع التعبير المنتظم r فمثلاً التعبير المنتظم a*i يتطابق مع سلاسل الحروف "a" و "aa" و "aaa" وهكذا كما يتطابق أيضا مع السلسلة الفارغة من الحروف Є ولتوضيح عملية التكرار بشكل أفضل سنقوم باجرائها على اللغات التي يمكن توليدها من إجراء نفس العملية على مجموعة من سلاسل الحروف وبفرض أن S هي مجموعة سلاسل حروف فإن:

S* = { Є } U S U SS U SSS U

أو بمعنى آخر فإن:

حيث Sn عبارة عن عملية الدمج لـ S S عدد n من المرات كما أن So = {Є}i وبالتالي فإن عملية التكرار يمكن تعريفها الآن كما يلي:

(L(r) = L(r

وعلى سبيل المثال فإن التعبير المنتظم *(a | bb) يتطابق مع أي من سلاسل الحروف Є و a و bb و aa و abb و bba و bbbb و aaa و aabb وهكذا وهو ما يمكن التعبير عنه كما يلي:

{L((a | bb)) = L (a | bb)* = {a | bb

{ ,Є , a, bb, aa, abb, bba, bbbb, aaa, aabb, abba} =

3.1.2 أولويات العمليات واستخدام الأقواس

في حالة وجود أكثر من عملية في التعبير المنتظم مثل *a | b فإنه لا بد من تحديد أولوية تنفيذ العمليات وهل تكون الأولوية لعملية الاختيار بين البدائل كما لو كانت *(a | b) أم تكون الأولوية لعملية التكرار بان تصبح i(*b) | a فبالطبع هناك فرق بين الحالتين كما يتضح مما يلي:

{,L((a | b)*) = { Є, a, b, aa, ab, ba, bb, aab, aba, baa

{ ,L (a | (b*)) = {Є, a, b, bb, bbb, bbbb, bbbbb

وعموماً فإن المعتاد بالنسبة لتنفيذ العمليات أن تكون أعلى أولوية لعملية التكرار ثم لعملية الدمج وأخيرا لعملية الاختيار بين البدائل وبالتالي فإنه بالنسبة للمثال السابق فإن التفسير الثاني هو السليم وبالنسبة للتعبير المنتظم *a | bc يتم تفسيره ليكون i((*b(c | a وأيضا التعبير المنتظم ab | c*d فإن تفسيره يكون (ab) | ((c*)d) وبصفة عامة فإنه عند الاحتياج لتنفيذ العمليات بأولويات غير الأولويات المعتادة فإنه يمكن استخدام الأقواس للقيام بهذا التغير. فمثلاً يتم استخدام الأقواس في التعبير المنتظم

i(a | b) c وذلك لكي يتم تنفيذ عملية الاختيار بين البدائل قبل عملية الدمج ونفس الحال بالنسبة للتعبير المنتظم *(a | bb) حيث يتم استخدام الأقواس لجعل عملية الاختيار بين البدائل تسبق عملية التكرار ويلاحظ مدى التشابه بين دور الأقواس في تغير أولويات تنفيذ العمليات بالنسبة للتعبيرات المنتظمة مع نفس الدور التي تقوم به في تغير أولويات تنفيذ العمليات الحسابية في التعبيرات الرياضية.
تسمية التعبيرات المنتظمة

غالباً ما يكون إعطاء أسماء للتعبيرات المنتظمة من الأمور المستحبة وذلك لكونه يبسط من شكل التعبير المنتظم لأن استخدام الاسم بدلاً من التعبير المنتظم ذاته وخاصة في التعبيرات المنتظمة المركبة يسهل من عملية كتابة واستخدام التعبير المنتظم فمثلاً عند إعداد تعبير منتظم للأعداد الصحيحة التي تتكون من سلسلة من الأرقام العشرية فإن التعبير المنتظم يكون كما يلي في صورته الأصلية:

9 | | 2 | 1 | 0) (9 | | 2 | 1 | 0)

ولكن عند إعطاء اسم للتعبير المنتظم الخاص بالأرقام العشرية وليكن كالآتي :

9 | | digit = 0 | 1 | 2

وذلك التعريف المنتظم للاسم digit يمكن من إعادة كتابة التعبير المنتظم السابق ليصبح كما يلي:

*digit digit

وبالرغم من أن استخدام التعريفات المنتظمة يبسط من شكل التعبير المنتظم ولكن تكمن المشكلة في صعوبة التفرقة بين اسم التعبير المنتظم وبين عملية الدمج لمجموعة الحروف التي يتكون منها هذا الاسم.

والآن نقدم مجموعة من الأمثلة الإضافية للتعبيرات المنتظمة حيث نقوم في كل مثال بإعطاء وصف نصي لسلاسل الحروف المطلوب تمثيلها قبل ترجمتها في شكل تعبير منتظم وهي المهمة التي غالبا ما تواجه الشخص الذي يقوم بكتابة المترجمات والذي يقع عليه عبء وصف مفردات اللغة الذي يقوم بإعداد المترجم الخاص بها حيث غالبا ما يتم هذا الوصف فى شكل منتظم كما هو الحال بالنسبة للتعبيرات المنتظمة.

مثال 1

بفرض أبجدية بسيطة تتكون من ثلاثة حروف فقط كالآتي :

{a, b, c} = ∑

وان مجموعة سلاسل الحروف التي يتم اشتقاقها من هذه الأبجدية تحتوي جميعاً على حرف b مرة واحدة فقط ففي هذه الحالة نجد أن هذه المجموعة يمكن وصفها بالتعبير المنتظم الآتي:

*(a | c)*b (a | c)

حيث يلاحظ أنه على الرغم من ظهور حرف الـ b في منتصف التعبير المنتظم فإن هذا الحرف ليس من الضروري أن يكون في منتصف سلاسل الحروف الذي يمكن اشتقاقها من هذا التعبير ولهذا فإن تكرار حرفي الـ a و الـ c قبل وبعد حرف الـ b يمكن أن يحدث بعدد مرات مختلف وجميع سلاسل الحروف التالية تتطابق مع التعبير المنتظم السابق:

b, abc, abaca, baaac, ccbca, cccccb

مثال 2

بفرض نفس الأبجدية التي في المثال السابق ولكن مجموعة سلاسل الحروف تحتوي على حرف الـ b مرة واحدة على الأكثر فعند ذلك فإن التعبير المنتظم لهذه المجموعة الأخيرة يمكن أن يتحقق باستخدام نفس الحل الخاص بالمثال السابق ولكن كأحد بديلين وهي الحالة التي يظهر فيها الحرف b لمرة واحدة أما في حالة عدم ظهور هذا الحرف فإن التعبير المنتظم *(a| c) يستخدم كبديل آخر وبالتالي فإن التعبير المنتظم الذي يوصف مجموعة سلاسل الحروف في هذا المثال سيكون كما يلي:

(a | c) | (a | c)* b (a | c)

ويمكن أيضا استخدام التعبير المنتظم التالي كحل آخر بديل:

(a| c) (b | є) (a | c)

وبالتالي فإن هذا المثال يقودنا إلى نقطة مهمة وهي أن نفس اللغة يمكن توليدها بواسطة عدة تعبيرات منتظمة مختلفة ولكننا عموماً نحاول بقدر الإمكان إيجاد أبسط التعبيرات المنتظمة التي توصف سلاسل الحروف التي تتكون منها اللغة.

مثال 3

بفرض مجموعة من سلال الحروف يتم تكوينها من الأبجدية { a, b } بحيث يظهر حرف الـ b في كل منها مرة واحدة على أن يحاط بحرف الـ a بعدد متساوي من المرات وذلك مثل سلاسل الحروف التالية:

,b, aba, aabaa, aaabaaa, aaaabaaaa

فهذه المجموعة لا يمكن وصفها باستخدام التعبيرات المنتظمة وذلك لان العملية الوحيدة للتكرار التي نملكها لا نستطيع بها تحديد عدد مرات التكرار وإذا استخدمنا التعبير المنتظم a*ba*i فإننا لا نستطيع أن نشترط أن عدد مرات تكرار حرف الـ a قبل حرف الـ b لا بد أن يتساوى مع عدد مرات تكراره بعد حرف الـ b ومن هذا المثال نستطيع استنتاج أن ليس جميع مجموعات سلاسل الحروف يمكن وصفها باستخدام التعبيرات المنتظمة ولغات البرمجة تحتوي على العديد من هذه المجموعات وعموما فإنه يطلق على مجموعة سلاسل الحروف التي يمكن وصفها في شكل تعبيرات منتظمة لقب المجموعة أو الفئة المنتظمة (Regular Set) وتعتبر مجموعة سلاسل الحروف غير منتظمة الموجودة في لغات البرمجة من أصعب المشاكل التي تواجه عملية تحليل مفردات هذه اللغات.

مثال 4

بفرض سلاسل من الحروف التي تتكون من الأبجدية { ∑ = {a,b,c وان هذه السلاسل لا تحتوي على حرفين متتاليين من حرف الـ b وبالتالي فإن أي حرفين من الـ b لا بد أن يظهر بينهما على الأقل أما حرف الـ a أو حرف الـ c ففي هذه الحالة نستطيع بناء التعبير المنتظم الذي يوصف تلك السلاسل من الحروف على عدة مراحل ولنبدأ بالجزء الذي يضمن ظهور حرفي الـ a أو الـ c بعد حرف الـ b وذلك باستخدام التعبير المنتظم التالي:

*((b (a | c)

وبعد ذلك نضيف له التعبير المنتظم *(a | c) الذي يتطابق مع الحالات التي لا يظهر فيها حرف الـ b ضمن سلسلة الحروف وبالتالي فإن التعبير المنتظم سيصبح كما يلي:

(((a | c)* | (b (a | c))

وحيث أن التعبير *(r* | s*i) يكافئ التعبير *(r | s) فإن التعبير المنتظم السابق يمكن تبسيطه ليصبح كالآتي:

*(((a | c) | (b (a | c))

وهو ما يمكن ايضا إعادة كتابته ليكون كالتالي:

*(a| c | ba | bc)

وهذا التعبير المنتظم الأخير يضمن عدم ظهور حرفين متتالين من حرف الـ b ولكنه يهمل حالة سلاسل الحروف التي تنتهي بحرف الـ b فمثلا لا يمكن اشتقاق سلاسل الحروف التالية من التعبير المنتظم الأخير:

,b, ab, cb, acb, cab, aab

على الرغم من أن هذه السلاسل لا تحتوي على حرف الـ b إلا مرة واحدة وبالتالي لا يوجد حرفان متتاليين من هذا الحرف ولكي نصحح ذلك نضيف للتعبير المنتظم السابق جزء اختياري في نهايته ليغطي هذه الحالات المشار إليها وليصبح التعبير المنتظم كما يلي:

(a | c | ba | bc)* (b |є)

مع ملاحظة أن التعبير الأخير يمكن أن يولد نفس سلاسل الحروف عن كتابته بالعكس وهو عندما يكون كالتالي:

*(b | є) (a | c | ab | cb)

ويمكن أيضا إعادة كتابته ليكون كمايلي:

(notb | b notb)* (b | є)

حيث تمت الاستعاضة عن الجزء a | c بالتسمية notb وذلك مثال لاستخدام الأسماء داخل التعبير المنتظم من اجل التبسيط وهو الإجراء المستحب وخاصة في حالة ما اذا كانت تحتوي فيه الأبجدية على عدد كبير من الحروف.

مثال 5

في هذا المثال الأخير سنقوم بالعكس وهو تقديم تعبير منتظم على أن نقوم بالتوصل إلى مجموعة سلاسل الحروف التي يمكن اشتقاقها من هذا التعبير وبفرض أن أبجدية هذه اللغة تشمل الحروف { a ,b , c} وأن التعبير المنتظم كان كالتالي:

(b | c) a (b | c)* a)* (b | c))

فهذا التعبير المنتظم يولد اللغة التي تتكون من سلاسل الحروف التي تحتوي على عدد زوجي من حرف الـ a سواء متتالية أو يفصل بينها حرفي الـ b أو الـ c ولتوضيح ذلك فإن الجزء الأيسر من هذا التعبير المنتظم والذي يتكون من الجزء التالي:

b | c)* a (b | c)* a)

يولد سلاسل الحروف التي تتكون من حرف الـ a مرتان فقط وأي عدد من حرفي الـ b أو الـ c على أن يظهر قبل أو بين حرفي الـ a وتكرار هذا الجزء يعطي جميع السلاسل التي تنتهي بحرف الـ a على أن يكون عدد المرات ظهور هذا الحرف زوجي أما الجزء الأخير فإنه يغطي الحالات التي لا تنتهي فيها سلاسل الحروف بحرف الـ a

2.2 إضافات التعبيرات المنتظمة

لقد تم في الجزء السابق تعريف التعبيرات المنتظمة التي تستخدم الحد الأدنى من العمليات حيث تم الاكتفاء بالعمليات الشائعة في التطبيقات المختلفة و اقتصر الأمر على استخدام ثلاثة عمليات فقط في جميع الأمثلة السابقة ولكننا قد لاحظنا أن استخدام هذه العمليات فقط يصعب من عملية إنشاء التعبيرات المنتظمة ويجعلها أكثر تعقيداً لهذا فسوف نقوم الآن بوصف بعض العمليات الإضافية الخاصة بالتعبيرات المنتظمة مع إعطاء الرمز الخاص بكل عملية جديدة مع الأخذ في الاعتبار أن بعض هذه العمليات ليس لها رمز محدد متفق عليه وفي هذه الحالات سوف نستخدم أكثر الرموز شيوعاً بالنسبة لكل عملية من تلك العمليات.

1.2.2 تكرار واحد أو أكثر

بفرض أن r هو احد التعبيرات المنتظمة فإن عملية تكرار هذا التعبير المنتظم والذي تكتب r*i تسمح بتكراره صفر أو أكثر من المرات بمعنى أن الحد الأدنى للتكرار في هذه العملية يكون صفر من

المرات ولكن في بعض الأحيان نحتاج إلى أن يكون الحد الأدنى للتكرار هو مرة واحدة أي أن يصبح عدد مرات التكرار مرة واحدة أو كثر من المرات ولا نسمح بسلسلة الحروف الفارغة ومثال على ذلك بفرض أننا نريد إعداد تعبير منتظم ليصف الأعداد الثنائية فإن التعبير المنتظم *(1 | 0)i سيسمح بوجود عدد ثنائي لا يحتوي على أي أرقام وهذا خطأ وحتى نتمكن من وصف الأعداد الثنائية بشكل صحيح لا بد أن يكون التعبير المنتظم كما يلي:

*(1 | 0)(1|0)

ولكون هذه الحالة تحدث في أحيانٍ كثيرة لذلك تم إضافة عملية التكرار التي يرمز لها بالرمز '+' والتي تعني أن التكرار يكون لمرة واحدة أو أكثر من المرات وبالتالي يمكن استخدامها مع التعبير المنتظم الذي يصف الإعداد الثنائية ليكون كالآتي:

+(1 | 0)

2.2.2 أي حرف

في حالات كثيرة نحتاج عند إعداد أحد التعبيرات المنتظمة إلى حدوث تطابق مع أي حرف من حروف أبجدية اللغة التي نريد وصفها بالتعبير المنتظم ففي هذه الحالة وبدون وجود رمز للتعبير عن أي حرف فإنه لا بد من ذكر جميع حروف الأبجدية في شكل بدائل متعددة ولكن عندما تكون عدد حروف الأبجدية كبير فإن ذلك يكون صعب إن لم يكن مستحيل لذلك فإن الرمز '.' يستخدم للدلالة على أي حرف من حروف الأبجدية فمثلا التعبير المنتظم الذي يصف مجموعة سلاسل الحروف التي يظهر فيها حرف الـ b مرة واحدة على الأقل يمكن كتابته كما يلي:

* b *

3.2.2 مدى الحروف

غالباً ما نحتاج إلى كتابة مدى من الحروف المتتالية عند إعداد التعبيرات المنتظمة مثل مجموعة الحروف الأبجدية الصغيرة من حرف الـ ' a ' إلى حرف الـ 'z' والتي نعبر عنها الآن باستخدام الشكل a | b | ... | z وكطريقة أخرى بديلة لذلك تستخدم الأقواس المربعة والشرطة للدلالة على ذلك فتكتب [a ⍰ z] وبنفس الطريقة فإن الشكل [Z ⍰ zA ⍰ a] يستخدم للدلالة عن جميع الحروف الأبجدية في اللغة الانجليزية والشكل [0 9] للدلالة على الأرقام العشرية وهكذا مع ملاحظة أن هذه الطريقة في التعبير عن مدى من الحروف يعتمد على كون الشفرة الخاصة بالحروف الأبجدية أو الأرقام في شفرة الـ ASCII تأتي بشكل متتابع بمعنى أن شفرة حرف الـ 'a' يليها شفرة حرف الـ 'b' ثم شفرة حرف الـ 'c' وهكذا.

4.2.2 أي حرف ما عدا مجموعة حروف

في بعض الأحيان نحتاج إلى استبعاد مجموعة من الحروف من ضمن بدائل حروف أبجدية في إحدى اللغات وللتعبير عن ذلك حالياً لا بد من حصر جميع حروف الأبجدية الخاصة باللغة مع استبعاد الحرف أو مجموعة الحروف التي لا نريد أن تتضمنها البدائل ولكن هذا الحصر يكون صعب عند زيادة عدد حروف الأبجدية ويستعاض عن ذلك باستخدام الرمز '~' الذي يعني النفي أو المكمل (Complement) لمجموعة البدائل فمثلاً التعبير المنتظم التالي:

(a | b | c) ~

يعني جميع أو أي حرف من الحروف التي تشتمل عليها أبجديات اللغة ما عدا حروف الـ 'a' و الـ 'b' و الـ 'c' ويمكن استخدام نفس الرمز مع حرف واحد وليس مجموعة حروف مستبعدة فمثلا b~ يعني أي حرف من حروف أبجدية اللغة ما عدا حرف الـ 'b' .

5.2.2 جزء اختياري

في حالات عديدة نريد إيضاح أن أحد أجزاء التعبير المنتظم اختياري بمعنى أن سلاسل الحروف التي يمكن توليدها من ذلك التعبير المنتظم قد تحتوي على هذا الجزء أو لا تحتوي عليه فمثلا لوصف الأعداد الصحيحة التي قد تحتوي على إشارة موجبة أو إشارة سالبة للدلالة على الأعداد الصحيحة الموجبة والسالبة ومع ذلك فإن العدد الصحيح قد لا يحتوي على أي إشارة وهو ما يدل أيضا على أن العدد الصحيح موجب وهذا ما يعني أن الإشارة اختيارية في العدد الصحيح ولوصف ذلك حالياً فإن التعبير المنتظم يكون كما يلي:

+[9 0] (- | +) | +[9 0]

وبدلاً من إعادة كتابة التعبير المنتظم بدون الجزء الاختياري فإن رمز علامة الاستفهام '?' يستخدم للدلالة عن الجزء الاختياري وبالتالي فإن التعبير المنتظم السابق يمكن إعادة كتابته ليصف الأعداد الصحيحة مع استخدام رمز الجزء الاختياري ليصبح التعبير المنتظم كما يلي:

+[9 0] ?(- | +)

. استخدام التعبيرات المنتظمة لوصف مفردات لغات البرمجة

إن مفردات أي لغة من لغات البرمجة يمكن تصنيفها في مجموعات محددة وهذه المجموعات غالباً لا تختلف من لغة برمجة إلى أخرى. فهناك مجموعة خاصة بالكلمات المحجوزة (Reserved Words) والتي تشمل على مجموعة الكلمات التي لها دلالة خاصة داخل لغة البرمجة مثل If و While وغيرها من الكلمات وكذلك هناك مجموعة الرموز الخاصة (Special Symbols) مثل الرموز الخاصة بالعمليات الرياضية وعمليات المقارنات والعمليات المنطقية وبعض هذه الرموز تتكون من حرف واحد مثل '+' و '*' و '<' و '=' والبعض الأخر تتكون من عدة حروف مثل ' ==' و '+ +' و '> =' وأيضا هناك مجموعة المعرفات (Identifiers) والتي تستخدم كأسماء للمتغيرات أو الوحدات البرمجية وغالبا ما تبدأ هذه المعرفات بأحد الحروف الأبجدية متبوعا بعدد من الحروف الأبجدية أو الأرقام أو كلاهما وأخيرا هناك مجموعة الثوابت (Constants) والتي تأخذ قيمة ثابتة سواء رقمية مثل

375 و 18.7 أو نصية مثل "Good Bye" أو حتى حرفية مثل 'A' أو '!' وسوف نقوم الآن بوصف كل مجموعة من المجموعات المختلفة لمفردات لغات البرمجة باستخدام التعبيرات المنتظمة:

1.3.2 الأعداد

الأعداد يمكن أن تكون أعداداً صحيحة (Integer Numbers) تتكون من سلسلة أرقام مسبوقة بإشارة أو بدون إشارة أو تكون أعداداً كسرية (Real Numbers) تحتوي على عدة أرقام قبل العلامة العشرية والأخرى بعد العلامة العشرية وقد تحتوي على إشارة وكذلك يمكن أن تكون الأعداد أعداد أسية (Exponential Numbers) تحتوي على حرف الـ 'e' أو حرف الـ 'E' متبوعاً بعدد صحيح ومسبوقاً بعدد كسري مثل iE-212.4 ويمكن الآن أن نكتب التعبير المنتظم الخاص بكل نوع من هذه الأعداد كما يلي:

+[9nat = [0

int = (+ | -)? nat

?(real = int ("." nat

?(number = int ("." nat)? ((e | E) int

حيث يلاحظ أن وصف الأعداد الكسرية real يحدد أن الجزء الخاص بالعلامة العشرية وما بعدها يعتبر جزء اختياري وذلك على اعتبار أن العدد الكسري يمكن أن يكون عدد صحيح فقط ولا يحتوي على كسر وأيضا الأعداد الأسية تعتبر هي التعريف الأشمل للأعداد Number لأنها قد تكون عدد صحيح أو عدد كسري فقط ولهذا فإن تعريفها يحتوي على بعض الأجزاء الاختيارية وأخيرا يلاحظ أن العلامة العشرية قد تم وضعها بين علامتي تنصيص "." وذلك لعدم الخلط بينها كعلامة عشرية وبين الرمز الخاص بإحدى عمليات التعبيرات المنتظمة التي تعني أي حرف من حروف أبجدية اللغة كما أوضحنا سلفاً.

2.3.2 الكلمات المحجوزة والمعرفات

تعتبر الكلمات المحجوزة من أسهل ما يمكن وصفه باستخدام التعبيرات المنتظمه وذلك لأن كل منها يتكون من سلسلة ثابتة من الحروف وبالتالي فإنه يمكن تجميع جميع الكلمات المحجوزة في تعبير منتظم واحد كمايلي:

| reserved = if | while | for

أما المعرفات فإن كل منها لا يتكون من سلسلة ثابتة من الحروف ولكن عادة ما يتكون المعرف من عدة حروف أبجدية وأرقام على أن يبدأ بأحد الحروف الأبجدية ويمكن أن يتم وصف المعرفات بإستخدام التعريفات المنتظمة التالية:

[letter = [a zA Z

[9digit = [0 ▯

*(identifier = letter (letter | digit

3.3.2 التعليقات

إن التعليقات (Comments) دائما ما يتم إهمالها عند مسحها بواسطة وحدة تحليل المفردات ولكن لكي يتم ذلك فإنه لابد من التعرف عليها أولا حتى يمكن استبعادها ولأن طريقة كتابة التعليقات قد تختلف من لغة إلى أخرى من لغات البرمجة لذلك فإن أسلوب التعرف عليها يتوقف على حسب الطريقة الخاصة بلغة البرمجة فمثلا في لغة الباسكال يكتب التعليق بالشكل التالي:

{this is a Pascal comment}

حيث يتم كتابة سلسلة من الحروف التي يتضمنها التعليق بين الحرفين المحددين { } بينما في لغة C تستخدم حروف محددة مختلفة حيث تأخذ الشكل التالي:

/* this is a C comment * /

وفي لغات أخرى يكون للتعليق حرف أو حرفان لتحديد بداية التعليق ولكن لا يوجد حرف محدد لنهاية التعليق على إفتراض أن التعليق ينتهي بنهاية السطر الذي يحتوي عليه التعليق وذلك مثل التعليقات التالية:

this is a prolog comment %

this is an Ada comment

وعموما فإن أعداد التعبير المنتظم الخاص بالتعرف على التعليقات يكون بسيطاً عندما تكون تعليقات لغة البرمجة لها محددات بداية ونهاية يتكون من حرف واحد أوتكون لها محدد بداية فقط حتى لوكان ذلك يتكون من عدة أحرف ولكن تكمن الصعوبة عندما تكون للتعليقات الخاصة بلغة البرمجة لها محددات بداية ونهاية تتكون من عدة حروف فمثلا التعبير المنتظم الخاص بتعليقات لغة الباسكال يكون كما يلي:

{ *({~) }

حيث يشترط ان يبدأ التعليق بالحرف المحدد للبداية وهو '{' وينتهي بالحرف المحدد للنهاية '}' وبينهما أي عدد من الحروف من أبجديات لغة الباسكال خلاف الحرف المحدد لنهاية التعليق الذي لابد الا يظهر في وسط التعليق ولكن مع نهايته فقط بينما التعبير المنتظم الخاص بتعليقات لغة البرلوج سيكون كما يلي:

*(newline ~) %

والتعبير المنتظم الخاص بتعليقات لغة الـ Ada يكون كالآتي:

*(newline ~)

حيث يفترض أن newline يتطابق مع الحرف الخاص بنهاية السطر والذي غالبا مايعبر عنه بالرمز 'i\n' في العديد من لغات البرمجة أما بالنسبة للتعليقات الخاصة بلغة السي فإن الصعوبة في إعداد تعبير منتظم في هذه الحالة تكمن في وجود محددات بداية ونهاية تتكون من أكثر من حرف حيث يستخدم الحرفان '*/' كمحدد بداية للتعليق والحرفان '/*' يستخدمان كمحدد نهاية للتعليق لذلك فإن التعرف على التعليقات الخاصة بلغة السي يتم بإسلوب مختلف وهذا سيتم تأجيله الأن وسوف نستعرض كيفية التعرف على التعليقات في مثل هذه الحالات في نهاية هذه الوحدة.
. الأتوماتة المنتهية

إن الأتوماتة المنتهية (Finite Automata) أو التي تسمى أحيانا بآلة الحالة المنتهية (-Finite State Machine) هي طريقة رياضية لوصف أنواع محددة من الخوارزميات ويمكن أيضا أن تستخدم في وصف عملية التعرف على الأنماط في سلاسل الحروف المدخلة ولذلك فأنها تستخدم في بناء الماسح الذي يقوم بتحليل المفردات ويوجد بالطبع علاقة قوية بين الأتوماتة المنتهية والتعبيرات المنتظمة وسوف نعرض في الجزء القادم كيفية بناء أتوماتة منتهية من تعبير منتظم وقبل أن نبدأ دراسة الأتوماتة المنتهية دعنا نفترض المثال التوضيحي التالي:

إن نمط المعرفات (Identifiers) الشائع تعريفه في لغات البرمجة يمكن أن يوصف بواسطة التعبير المنتظم التالي:

*(identifiers = letter (letter | digit

وذلك بفرض أن الأسماء المنتظمة letter و digit قد سبق تعريفها وبالتالي فإن هذا التعبير المنتظم السابق يمثل سلسلة الحروف التي تبدأ بحرف أبجدي ثم تتابع بأي عدد من الحروف الأبجدية و الأرقام العشرية وعملية التعرف على مثل هذه السلسله من الحروف يمكن وصفها بإستخدام مخطط الإنتقالات (Transitions Diagram) التالي:

والدائرتان المرقمتان بالرقم (1) والرقم (2) في هذ ا المخطط يمثلان الحالات ((States أو المراحل التي تمر عليها عملية التعرف حيث تسجل مختلف الأنماط التي يمكن أن نراها بينما تعبر الأسهم عن الانتقالات (Transitions) والتي تسجل التغير من حالة لأخرى نتيجة لحدوث التطابق مع الحرف أو مع مجموعة الحروف الذي يعنونه كل سهم وفي المخطط السابق تعتبر الحالة رقم (1) هي حالة البداية (Start State) وهي الحالة التي تبدأ عندها عملية التعرف على النمط ويتم تمييز تلك الحالة في المخطط من خلال سهم غير معنون ينتهي عند هذه الحالة وبدون وجود حالة سابقة يبدأ منها أما الحالة رقم (2) فهي تمثل الوضع عند وجود تطابق مع حرف أبجدي والذي يعبر عن الإنتقال الممثل بالسهم المتجه من الحالة رقم (1) إلى الحالة رقم (2) والمعنون بـ letter وبمجرد الوصول إلى الحالة رقم (2) فإن أي عدد من الحروف الأبجدية أو الأرقام يمكن أن نراها وعند حدوث التطابق مع أي منها نعود إلى الحالة رقم (2) مرة أخرى وعموما فإن الحالة أوالحالات التي تمثل نهاية عملية التعرف على النمط والتي يعني الوصول إليها نجاح تلك العملية تسمى بحالات القبول (Accepting States) ويتم تمييزها

في مخطط الإنتقالات باستخدام الدائرة المزدوجة مثل الحالة رقم (2) في المخطط السابق والتي تعتبر حالة مقبولة أو منتهية وذلك لأن حدوث التطابق مع حرف أبجدي واحد أو مع حرف أبجدي يتبعه أي عدد من الحروف الأبجدية أو الأرقام العشرية يصل بنا إلى هذه الحالة وهومايعتبر معرف صحيح أو مقبول في لغة البرمجة.

وعملية التعرف على سلسلة حروف محددة تمثل أحد المعرفات يمكن الأن التعبيرعنها بواسطة قائمة الحالات المتعاقبة و الأنتقالات من خلال المخطط الذي يستخدم في عملية التعرف على سلسلة الحروف فمثلا عملية التعرف على أو تمييز سلسلة الحروف Count كأحد المعرفات يمكن التعبير عنها كمايلي:

حيث تم عنونة كل إنتقال بالحرف الأبجدي الذي يحدث التطابق في كل خطوة .

1.3 الأتوماتة المنتهية الحتمية

إن مخططاً مثل المخطط الذي تم شرحه سابقا يستخدم بفاعلية لوصف الأتوماتات المنتهية وذلك لكونه يسمح لنا بمشاهدة أداء الخوارزميات بسهولة ولكن مع ذلك فإنه من الضروري وجود وصف رسمي (Formal Description) للأتوماتة المنتهية لهذا سنقوم الأن بتقديم تعريف رياضي لها ولكن لن نحتاج لمثل هذا التعريف في أغلب الأوقات ونعتمد فقط على المخططات لوصف أغلب الأمثلة الخاصة بالأتوماتة المنتهية ويجب أن نلاحظ أيضا أننا سنقوم فقط بوصف الأتوماتة المنتهية الحتمية (Deterministic Finite Automata) والتي يطلق عليها إختصارا DFA وهي الاتوماتة التي يتم فيها تحديد الحالة التالية مع كل إنتقال حسب الحالة الحالية وحرف المدخلات الحالي فقط ويعتبرهذا النوع من الأتوماتة المنتهية هو حالة خاصة من الأتوماتة المنتهية غير الحتمية (Nondeterministic Finite Automata) وهي الاتوماتة التي سيتم دراستها لاحقا .

والأن نقدم مجموعة من أمثلة الأتوماتية المنتهية الحتمية توازي الأمثلة السابق تقديمها مع التعبيرات المنتظمة:

مثال 6:

مجموعة سلاسل الحروف التي تحتوي على حرف ألـ b مرة واحدة فقط يمكن قبولها بإستخدام الأتوماتة المنتهية الحتمية التالية:

مثال 7 :

مجموعة سلاسل الحروف التي تحتوي على حرف الـ b مرة واحدة على الأكثر يمكن قبولها باستخدام الأتوماتة المنتهية الحتمية التالية:

يلاحظ أن مخطط الإنتقالات في هذا المثال هونفس مخطط الإنتقالات الخاص بالمثال السابق ولكن مع إجراء تعديل بسيط وهو جعل حالة البداية رقم (1) إحدى الحالات المقبولة في المخطط الأخير.

مثال 8 :

في الجزء السابق قدمنا التعريفات المنتظمة للأعداد الثابتة في شكلها العلمي كما يلي:

+[9nat = [0 ▯

int = (+ | -)? nat

?(real = int ("." nat

?(number = int ("." nat)? ((e | E) int

ولإعداد DFA لسلسلة الحروف التي تتطابق مع هذه التعريفات فإنه من الأفضل إعادة كتابة تلك التعريفات المنتظمة لتصبح كما يلي :

[9digit = [0 ▯

+nat = digit

int = (+ | -)? nat

?(real = int ("." nat

?(number = int ("." nat)? ((e | E) int

والآن من السهل إعداد DFA للأعداد الطبيعية (nat) والذي يكون كالآتي:

وسيتم تأجيل ترقيم أو إعطاء أسماء للحالات التي يتضمنها مخطط الإنتقالات إلى حين الأنتهاء من إعداد الأتوماتة المنتهية الحتمية الكاملة للأعداد الثابتة.

وبالنسبة للأعداد الصحيحة (int) فإن الأتوماتة المنتهية الحتمية الخاصة بها يزداد قليلاً في صعوبته وذلك بسبب الجزء الإختياري الخاص بالإشارة حيث نلاحظ أن العدد الصحيح يمكن أن يبدأ برقم عشري أو بإشارة ورقم عشري وبالتالي فإن مخطط الإنتقالات سيصبح كالآتي:

ويمكن بسهولة إضافة الجزء الإختياري الخاص بالكسر كما يلي:

ويلاحظ أن مخطط الإنتقالات السابق يحتوي على حالتين مقبولتين وذلك ليعكس أن الجزء الخاص بالكسر جزء إختياري. وفي النهاية نحتاج لاضافة الجزء الإختياري الآخر والخاص بالجزء الأسي في الإعداد وللقيام بذلك يجب ملاحظة أن هذا الجزء يجب أن يبدأ بحرف E أو حرف e ويمكن أن يحدث فقط بعد الوصول لأحد الحالتين المقبولتين وسيكون مخطط الإنتقالات الذي يصف الأتوماتة المنتهية الحتمية الخاصة بالتعرف على الأعداد كما يلي:

مثال 9 :

التعليقات التي توجد في أغلب لغات البرمجة يمكن أيضا إعداد الأتوماتة المنتهية الحتمية للتعرف عليها وسنكتفي في هذا الجزء بالتعليقات التي لها محددات بداية ونهاية يتكون من حرف واحد مثل تعليقات لغة الباسكال الذي تستخدم الأقواس { } كمحددات حيث يتم كتابة سلسلة الحروف التي يتضمنها التعليق بينهما ويمكن وصف الأتوماتة المنتهية الحتمية الخاصة بالتعرف على تعليقات لغة الباسكال باستخدام مخطط الإنتقالات التالي:

وكلمة other في المخطط السابق يعني جميع الحروف فيما عدا الحرف ' }' الخاص بنهاية التعليق .

ألآن وبعد الإنتهاء من الأمثلة السابقة يجب ملاحظة أن مخطط الإنتقالات الخاص بأي أتوماتة منتهية حتمية لا يقوم بتمثيل كل الأشياء التي يحتاجها ولكن يعطي فقط إطار عام لعملياته وليس فقط ذلك وإنما نجد أيضا أن التعريف الرياضي يحدد أن أي أتوماتة منتهية حتمية يجب أن يحتوي على إنتقال خاص لكل حالة من مجموعة حالاته مع كل حرف من حروف أبجديته بينما مخطط الإنتقالات الخاص به لا يظهر جميع تلك الإنتقالات بما فيها الإنتقالات التي تنتج عن حدوث أخطاء في حالة ظهور حرف لايمكن ظهوره مع حاله محدده من حالات الأتوماتة المنتهية الحتمية ومع ذلك ومن ناحية أخرى نجد أن التعريف الرياضي لا يقوم بوصف سلوك الخوارزمية الخاصة بالأتوماتة المنتهية الحتمية في المواقف المختلفة فمثلا لا يوضح التعريف الرياضي ما الذي يحدث عند حدوث خطأ وكذلك لايحدد ما الإجراء الذي يتم عند الوصول لأحد الحالات المقبولة أو عند حدوث التطابق مع أحد الحروف أثناء الإنتقال من حالة إلى أخرى والإجراء الفعلي الذي يتم عند حدوث هذا الموقف الأخير هوأن يتم نقل هذا الحرف من سلسلة حروف المدخلات إلى سلسلة الحروف المجمعة و التي تنتمي لمفردة محددة بينما الإجراء الذي يتم عند الوصول لأحد الحالات المقبولة هو أن يتم إرجاع المفردة التي تم التعرف عليها مع جميع الخصائص المرتبطة بها أما بالنسبة لما يتم عند الوصول لحالة الخطأ فإما أن يتم إرجاع جميع الحروف السابق مسحها لمحاولة التعرف عليها ضمن النمط الخاص بمفردة أخرى أو أن يتم إظهار رسالة خطأ.

ومن ناحية أخرى فإننا نوجه إهتمامنا الآن إلى كيفية الوصول إلى حالة البداية في أي أتوماتة حتمية فمن المعروف أن جميع لغات البرمجة تتضمن عدة مفردات وكل مفردة من هذه المفردات يتم التعرف عليها بواسطة الأتوماتة المنتهية الحتمية الخاصة بها وإذا كانت كل مفردة تبدأ بحرف مختلف عن الأخرى فعند ذلك يكون الأمر سهل لانه في هذه الحاله يمكن دمج حالات البداية الخاصة بتلك المفردات

في حالة بداية واحدة وبالتالي تجميع الأتوماتة المنتهية الحتمية المختلفة الخاصة بها في أتوماتة منتهية حتمية واحدة مجمعة فمثلا بفرض المفردات التي تكون سلسلة الحروف التي تمثلها من '=' و '=>' و ':=' وكانت الأتوماتة المنتهية الحتمية الخاصة بالتعرف على كل منها تأخذ الشكل التالي على التوالي:

وحيث أن كل مفردة من تلك المفردات تبدأ بحرف مختلف فإنه يمكن توحيد الأتوماتة المنتهية الحتمية الخاصة بكل منها في الأتوماتة المنتهية الحتمية التالية:
وتكمن المشكلة إذا كانت هناك عدة مفردات تبدأ بنفس الحرف فمثلا المفردات '>' و '=>' و '<>' تتشابه جميعا في حرف البداية وبالتالي لا يمكن إعداد أتوماتة منتهية حتمية مجمعة لها وذلك لأنه يجب أن يكون هناك إنتقال وحيد من كل حالة من حالات الأتوماتة المنتهية الحتمية مع كل حرف من حروف أبجديتها وهذا لم يحدث في مخطط الإنتقالات المجمع التالي:

وبدلا من ذلك يجب ترتيب الأتوماتة المنتهية الحتمية المجمعة بحيث يكون هناك إنتقال وحيد يحدث في كل حالة بحيث يصبح مخطط الإنتقالات كالآتي:

قاعدة عامة يجب أن تكون هناك قدرة على دمج جميع المفردات في أتوماتة منتهية حتمية واحدة مشتركة بهذا الأسلوب السابق ولكن تعقيد مثل هذه المهمة قد يصبح كبيرا وخاصة إذا ما تم بطريقة غير منظمة وحل هذه المشكلة يمكن أن يحدث بتوسيع التعريف الخاص بالأتوماته المنتهية ليشمل إحتمال وجود أكثر من إنتقال واحد من إحدى الحالات مع بعض الحروف وهذا النوع الجديد من الأتوماتة المنتهية تسمى أتوماتة منتهية غير حتمية (Nondeterministic Finite Automata) و ما يطلق عليها اختصاراً NFA وفي نفس الوقت يجب تطوير طريقة منظمة لتحويل أي أتوماتة منتهية غير حتمية إلى أتوماتة منتهية حتمية. وهذا ما سيتم دراسته في الجزء التالي .
الأتوماتة المنتهية غير الحتمية

الآن وقبل التعرض للتعريف الرياضي للاتوماته المنتهية غير الحتمية نحتاج للقيام بتعميم خاص بالانتقالات يكون مفيداً في التعامل مع هذا النوع وهو مفهوم الانتقال الفارغ (Є-transition) الذي يتم دون استهلاك أي حرف من سلسلة حروف المدخلات كما لو كان يحدث تطابق مع سلسلة الحروف الفارغة التي نرمز لها بالرمز Є ويظهر في مخطط الانتقالات كما يلي:

قد يعتبر الانتقال الفارغ إلى حد ما غير بديهي لكونه يحدث بشكل عفوي ودون تحرك المؤشر أو حدوث تغيير في سلسلة حروف المدخلات ولكنه فعال جداً لسببين الأول أنه يمكن أن يعبر عن الاختيار بين البدائل بطريقة لا تحتاج إلى دمج حالات الاتوماته فمثلا عند الاختيار بين المفردات '==' و '=>' و '=' يمكن التعبير عن ذلك من خلال دمج الاتوماته الخاصة بكل منها كما يلي :
حيث يتميز المخطط السابق انه قام بدمج الاتوماته الخاصة بكل مفردة دون إجراء أي تغير على أي منهم ودون دمج حالة البداية الخاصة بهم وإنما تمت إضافة حاله بداية جديدة لدمجهم معاً . أما الميزة أو السبب الثاني لاستخدام الانتقال الفارغ كونه يمكن من وصف التطابق مع سلسلة الحروف الفارغة بشكل واضح كما يلي:

وذلك للتعبير عن حدوث القبول دون وجود تطابق مع أي حرف بدلا من استخدام الأتوماتة المنتهية الحتمية التالية التي تعبر عن نفس الموقف ولكن بشكل غير واضح:

وبالنسبة للاتوماته غير الحتمية، فهي تتشابه مع الاتوماته الحتمية، ما عدا أننا ووفقاً للشرح السابق نحتاج إلى توسعة حروف الأبجدية ∑ لتشمل الحرف الفارغ ϵ وأيضا نحتاج إلى إعادة تعريف دالة الانتقال T لتسمح عند حدوث التطابق مع أحد الحروف الانتقال لأكثر من حالة واحدة وذلك يتم بجعل ناتج هذه الدالة مجموعة حالات فمثلاً في المخطط التالي:

نجد انه من الحالة (1) يمكن الانتقال لكل من الحالة (2) أو الحالة (3) مع حرف المدخلات '<' وهو ما نعبر عنه باستخدام دالة الانتقال T كما يلي :

{T(1, <) = {2, 3

وهو ما يعني ان ناتج هذه الدالة هو مجموعة حالات وبالتالي فإن مدى دالة الانتقال أصبح عبارة عن مجموعة المجموعات الجزئية للحالات والتي تسمى بمجموعة القوة (Power-Set)) والتي تتكون من جميع المجموعات الجزئية (Subsets) من مجموعة حالات الاتوماته حيث يرمز لمجموعة القوة لمجموعة الحالات S بالرمز ρ(S)i ونقوم الان بتعريف الاتوماته المنتهية غير الحتمية (NFA) .

من التعريف السابق يجب ملاحظة أن مجموعة الحالات المتتابعة SN , , S0 , S1 , S2 التي يتم اختيارها من مجموعة الحالات S باستخدام دالة الانتقال T وسلسلة حروف المدخلات ليس من الضروري أن يتم تحديدها بشكل وحيد بل من الممكن أن يكون هناك أكثر من مجموعة من الحالات المتتابعة لنفس سلسلة حروف المدخلات ولهذا السبب يطلق على هذا النوع من الاتوماته لقب اتوماته غير حتمية فسلسلة الانتقالات التي تقبل سلسلة محددة من الحروف لايتم تحديدها في كل خطوة حسب الحالة الحالية وحرف المدخلات التالي بل من الممكن في أي وقت ظهور أي عدد من الحروف الفارغة ϵ ضمن سلسلة الحروف والذي يقابلها مجموعة من الانتقالات الفارغة في الأتوماتة المنتهية غير الحتمية ولهذا فإن الأتوماتة المنتهية غير الحتمية لا يمكن أن تمثل خوارزمية بالمعنى الصحيح ولكن يمكن أن نقول أنه يمكن محاكاتها باستخدام خوارزمية وهو ما سيتضح في الجزء التالي بينما نقدم الآن بعض الأمثلة الخاصة بالأتوماتة المنتهية الحتمية.

مثال 10

بفرض المخطط التالي الخاص بإحدى الأتوماتة المنتهية غير الحتمية :

فوفقاً لهذا المخطط فإن سلسلة الحروف "abb" يمكن قبولها بواسطة أي من سلسلتي الانتقالات التالية:

حيث نلاحظ أن الانتقال من الحالة (1) إلى الحالة (2) يتم مع ظهور الحرف 'a' بينما الانتقال من الحالة (2) إلى الحالة (4) يتم مع ظهور الحرف 'b' مما يسمح من قبول سلسلة الحروف "ab" ثم باستخدام الانتقال الفارغ من الحالة (4) إلى الحالة (2) يمكن قبول جميع سلاسل الحروف التي تتطابق مع التعبير المنتظم +ab وكذلك فإن الانتقال من الحالة (1) إلى الحالة (3) يتم مع ظهور الحرف 'a' والانتقال من الحالة (3) إلى الحالة (4) يتم مع ظهور الحرف الفارغ ϵ مما يمكن من قبول جميع سلاسل الحروف التي تتطابق مع التعبير المنتظم *ab وأخيراً فإن استخدام الانتقال الفارغ من الحالة (1) إلى الحالة (4) يسمح بقبول سلسلة الحروف التي تتطابق مع التعبير المنتظم*b

وبالتالي فإن الأتوماتة المنتهية غير الحتمية السابقة يقبل نفس اللغة المقبولة بواسطة التعبير المنتظم ab+ والذي يمكن تبسيطه ليصبح | ab* | b*i

*a | Є) B)

وهو ما يمكن التعرف عليه باستخدام الـ DFA التالية :

مثال 11

بفرض الـ NFA التالية :

فإنه يقبل سلسلة الحروف "acab" عن طريق سلسلة الانتقالات التالية :
وفي الحقيقة ليس من الصعب أن نلاحظ أن الأتوماتة المنتهية غير الحتمية السابقة تقبل نفس اللغة المقبولة بواسطة التعبير المنتظم i(a | c)* b
3.3 إعداد البرنامج الخاص بالأتوماتة المنتهية

هناك عدة طرق لتحويل أيا من الأتوماتة المنتهية الحتمية أو الأتوماتة المنتهية غير الحتمية إلى برنامج مكتوب وسوف نستعرض تلك الطرق في هذا الجزء مع الأخذ في الاعتبار أن جميع الطرق ليست صالحة للاستخدام مع مرحلة تحليل المفردات لهذا سنركز في آخر جزئيين من هذه الوحدة على الطرق المناسبة لهذه المرحلة من المترجم بشيء من التفصيل والآن نعود إلى مثال الأتوماتة المنتهية الحتمية التي تقبل المعرفات التي تتكون من حرف أبجدي متبوع بسلسلة من الحروف الأبجدية أو الأرقام العشرية أو كلاهما والتي يأخذ مخطط الانتقالات الخاص به الشكل التالي:

فإن أول وأسهل طريقة لمحاكاة هذه الأتوماتة المنتهية الحتمية هو كتابة البرنامج على الشكل التالي :

```
{ starting in state 1 }

if the next character is a letter then

advance the input

{ now in state 2 }

while the next character is a letter or a digit do

{ advance the input { stay in state 2

end while
```

{ go to state 3 without advancing the input }

accept

else

{ error or other cases }

end if

فالطريقة السابقة يتم فيها معالجة حالات الأتوماتة بشكل ضمني وليس صريحاً. وهذا ما يوضحه استخدام التعليقات لتحديد الحالة التي نقف عندها مع كل انتقال داخل الاتوماته وهذا مقبول إذا ما كان عدد الحالات صغير وإذا ما كان التكرار في الأتوماتة المنتهية الحتمية محدد ولهذا فإن هذه الطريقة تكون مناسبة ويمكن استخدامها في كتابة الماسحات الصغيرة ولكن هناك عيبان أساسيان في هذه الطريقة الأول إن كل أتوماتة منتهية حتمية يتم التعامل معها بشكل مختلف وهو ما يجعل من الصعب إعداد خوارزمية عامة لتحويل أي أتوماتة حتمية إلى برنامج مكتوب باستخدام تلك الطريقة. أما العيب الثاني فيتمثل في أن تعقيد البرنامج سيزداد بشكل كبير مع زيادة عدد حالات الاتوماته وزيادة عدد الانتقالات بينها ولتأكيد ذلك بفرض الأتوماتة المنتهية الحتمية التي تم عرضها في المثال رقم (9) السابق والخاص بالتعرف على التعليقات المستخدمة في لغة السي والمطلوب إعداد البرنامج الخاص بهذه الاتوماته وسوف نترك للقارئ القيام بذلك وسيلاحظ مدى الزيادة في التعقيد في هذا البرنامج نتيجة للزيادة البسيطة في عدد الحالات والانتقالات وكبديل للطريقة السابقة هناك طريقة أخرى أفضل تقوم باستخدام احد المتغيرات لمتابعة الحالة الحالية مع تنفيذ الانتقالات من خلال اثنين من أوامر التفرع المتداخلة داخل الجزء الخاص بالتكرار حيث يقوم أمر التفرع الأول باختبار الحالة الحالية بينما يقوم أمر التفرع الثاني باختبار حرف المدخلات وعلى سبيل المثال فإن الأتوماتة المنتهية الحتمية السابقة الخاصة بالتعرف على المعرفات يمكن تحويلها إلى البرنامج التالي وفقاً لهذه الطريقة الأخيرة :

{ state: = 1 { start

while state = 1 or 2 do

case state of

case input character of :1

letter: advance the input

state : = 2

{ error or other } = : else state

end case

case input character of : 2

letter , digit : advance the input

state : = 2

else state : = 3

end case

end case

end while

if state = 3 then accept else error

نلاحظ كيف أن البرنامج وفقا لهذه الطريقة يعكس الأتوماتة المنتهية الحتمية مباشرة ومع كل انتقال يتم تغيير قيمة المتغير state برقم الحالة التي نصل عندها مع تقديم مؤشر المدخلات حرفاً للأمام (باستثناء الانتقــال من الحالــة رقم (2) إلى الحالة رقم (3) ولتوضيح أن برنامج تلك الطريقة سيزداد حجماً ولكن لن يزداد تعقيداً مع زيادة عدد الحالات الخاصة بالأتوماتة المنتهية الحتمية نقوم الآن بإعداد البرنامج المقابل للأتوماتة المنتهية الحتمية التي تتعرف على التعليقات الخاصة بلغة السي والذي يكون كالآتي:

state: = 1

while state = 1, 2, 3 or 4 do

case state of

case input character of : 1

advance the input :"/"

state := 2

{ error or other} = : else state

end case

case input character of : 2

advance the input : " *"

state := 3

{ error or other } = : else state

end case

case input character of : 3

advance the input :"*"

state := 4

{ else advance the input { stay in state 3

end case

case input character of : 4

advance the input : "/"

state := 5

{advance the input { stay in state 4 : "*"

else advance the input

state : = 3

end case

end case

end while

if state = 5 then accept else error

بالإضافة للطريقتين السابقتين لإعداد البرنامج المقابل للأتوماتة المنتهية الحتمية توجد طريقة أخرى يتم فيها التعبير عن الاتوماته من خلال إحدى تراكيب البيانات التي يطلق عليها جدول الانتقالات (Transition Table) ثم بعد ذلك يتم إعداد برنامج عام يتفاعل مع الجدول ويصلح لأي أتوماتة منتهية حتمية لان الذي يتغير فقط هو محتويات جدول الانتقالات الذي يأخذ شكل مصفوفة ثنائية الأبعاد ومفهرسة بحالات وحروف مدخلات الأتوماتة المنتهية الحتمية حيث يعبر الجدول عن قيم دالة الانتقالات الخاصة بتلك الأتوماتة المنتهية الحتمية. فمثلا جدول الانتقالات التالي يمثل الأتوماتة المنتهية الحتمية التي تتعرف على المعرفات:

وتتضمن خانات الجدول الانتقالات التي تظهر في الـ DFA فقط أما باقي الخانات فتترك فارغة ونفترض أن أول حالة تظهر في الجدول هي دائماً حالة البداية في الـ DFA ولكن لا يوضح الجدول الحالات المقبولة وكذلك لا يحدد أي الانتقالات التي يصاحبها تحريك مؤشر حروف المدخلات وهذه المعلومات يمكن إضافتها لجدول الانتقالات أو أن تشملها تركيبة بيانات منفصلة فإذا أضفنا تلك المعلومات للجدول السابق فإنه يمكن إضافة عمود في النهاية يوضح ما إذا كانت كل حالة من الحالات التي يتضمنها الجدول أحد الحالات المقبولة واستخدام الأقواس للتعبير عن عدم تحريك مؤشر المدخلات فإن هذا الجدول سيصبح كما يلي:

أما بالنسبة للـ DFA الخاص بالتعرف على تعليقات لغة السي فإن جدول الانتقالات الخاص بها سيكون كما يلي :

والآن نستطيع إعداد البرنامج العام القادر على التفاعل مع أي جدول انتقالات خاص بأي أتوماتة منتهية حتمية وذلك بفرض أن الانتقالات سيتم الاحتفاظ بها في جدول الانتقالات T وان يتم التعبير عن تحرك أو عدم تحرك مؤشر المدخلات بشكل منفصل من خلال المصفوفة المنطقية Advance المفهرسة أيضا بالحالات وحروف المدخلات اما الحالات المقبولة فسيتم تحديدها باستخدام المتجه المنطقي Accept المفهرس بالحالات وفيما يلي خطوات هذ البرنامج العام:

```
state: = 1
ch := next input character
while not Accept [ state ] do
[newstate : = T[state, ch
if Advance[state, ch] then
ch := next input character
state := newstate
end while
```

if Accept[state] then accept

وهذه الطريقة الأخيرة تتميز بصغر حجم البرنامج وكذلك إمكانية استخدام نفس البرنامج مع العديد من الأتوماتة المنتهية الحتمية المختلفة ولكن تكمن المشكلة في زيادة حجم جدول الانتقالات مع زيادة عدد الحالات وحروف المدخلات التي تتضمنها الأتوماتة المنتهية الحتمية.

وفي النهاية يجب توضيح أن الأتوماتة المنتهية غير الحتمية يمكن من إعداد البرنامج الخاص بها بنفس الطرق المستخدمة مع الأتوماتة المنتهية الحتمية ولكن لكون الأتوماتة المنتهية غير الحتمية غير حتمية، فإنه يوجد عدة سلاسل مختلفة من الانتقالات لا بد من اختبارها. لذلك فإن البرنامج الذي يحاكي الأتوماتة المنتهية غير الحتمية لا بد أن يسجل جميع الانتقالات الأخرى ولان ذلك يجعل من مثل هذه البرامج بطيئة وغير فعالة لهذا فبدلاً من إعداد برنامج لتمثيل الأتوماتة المنتهية غير الحتمية فإنه من المفضل أن يتم تحويلها إلى أتوماتة منتهية حتمية وإعداد برنامج أكثر كفاءة وفعالية للأخيرة وهذا ما سوف نقدمه في الجزء التالي.
. تحويل التعبيرات المنتظمة إلى أتوماتة منتهية حتمية

في هذا الجزء سندرس كيفية تحويل التعبيرات المنتظمة إلى أتوماتة منتهية حتمية والعكس مما يعني أن الاثنان متماثلان ولكن بسبب أن التعبيرات المنتظمة مضغوطة وشديدة التركيز لذلك فنحن نفضل التعامل مع الاتوماته عن التعبيرات المنتظمة بينما بالنسبة للماسح الخاص بتحليل المفردات فإنه غالباً ما يبدأ بالتعامل مع التعبيرات المنتظمة قبل الشروع في بناء الاتوماته المنتهية الحتمية ولهذا السبب نهتم بالتحويل من التعبيرات المنتظمة إلى اتوماته منتهية حتمية وليس العكس واسهل طريقة لتحويل التعبير المنتظم إلى أتوماتة منتهية حتمية أن يتم ذلك من خلال مرحلة وسيطة وهذه المرحلة هي أتوماتة منتهية غير حتمية بمعنى أن نبدأ بتحويل التعبير المنتظم إلى أتوماتة منتهية غير حتمية ثم بعد ذلك يتم التحويل من أتوماتة منتهية غير حتمية إلى أتوماتة منتهية حتمية . وبالرغم من وجود طريقة تمكن من تحويل التعبير المنتظم إلى أتوماتة منتهية حتمية مباشرة، ولكن لشدة تعقيد هذه الطريقة فإنه يفضل استخدام الطريقة السابقة التي تتم على مرحلتين عند استخدام الطريقة المباشرة. وبعد الوصول إلى أتوماتة منتهية حتمية نقوم باستخدام الأسلوب الذي قدمناه في الجزء السابق لبناء البرنامج المقابل لتلك الاتوماته المنتهية ومن ذلك نستنتج أن الماسح الخاص بتحليل المفردات يتم بناءه على ثلاثة خطوات متعاقبة كالموضحة في الشكل التالي :

1.4 التحويل من تعبير منتظم إلى NFA

إن كل عملية من عمليات التعبير المنتظم يتم تحويلها بشكل منفصل إلى الأتوماتة المنتهية غير الحتمية المقابلة لها ثم بعد ذلك يتم تجميع تلك الأجزء المنفصلة من الأتوماتة المنتهية غير الحتمية الناتجة من التحويل ودمجها في أتوماتة منتهية حتمية واحدة باستخدام الانتقال الفارغ ويصبح الناتج هو الأتوماتة المنتهية غير الحتمية المجمعة التي تقابل التعبير المنتظم الأصلي.

1.1.4 التعبيرات المنتظمة البسيطة

التعبيرات المنتظمة البسيطة تأخذ الشكل a أو Є أو Ø حيث تمثل الـ a أحد حروف الأبجدية بينما تمثل الـ Є سلسلة الحروف الفارغة أما الـ Ø فلا تمثل أي سلسلة حروف وفي حالة التعبير المنتظم الذي يتكون من حرف الـ a فإن الـ NFA التي تقابله تأخذ الشكل التالي :

وبنفس الطريقة فإن الـ NFA المقابلة للتعبير المنتظم التي يتضمن الرمز Є تكون كما يلي:

2.1.4. الـدمـــــج

عند بناء الأتوماتة المنتهية غير الحتمية التي تقابل التعبير المنتظم rs حيث كل من r و s يمثلان تعبيران منتظمان في حد ذاتها وبفرض أن الأتوماتة المنتهية غير الحتمية الخاصة بكلاهما قد تم بناءهما ويأخذ كلاهما الشكلان التاليان

وذلك بافتراض أن كلا منهما يتضمن على حالة مقبولة واحدة فعند ذلك فإن الأتوماتة المنتهية غير الحتمية المقابلة للتعبير المنتظم rs ستكون كالتالي:
حيث تم توصيل الحالة المقبولة للأتوماتة المنتهية غير الحتمية المقابلة للتعبير المنتظم r مع حالة البداية الخاصة بالـ الأتوماتة المنتهية غير الحتمية المقابلة للتعبير المنتظم s باستخدام الانتقال الفارغ وأصبحت حالة البداية الخاصة بالأتوماتة المنتهية غير الحتمية الأولى هي حالة البداية للأتوماتة المنتهية غير الحتمية المجمعة بينما الحالة المقبولة للأتوماتة المنتهية غير الحتمية الثانية فأصبحت هي الحالة المقبولة للأتوماتة المنتهية غير الحتمية المجمعة.

3.1.4 الاختيار بين البدائل

لبناء الأتوماتة المنتهية غير الحتمية تقابل التعبير المنتظم r | s وتحت نفس الفروض السابقة فإنها ستكون كما يلي :

حيث تمت إضافة حالة بداية جديدة وحالة مقبولة جديدة للأتوماتة المنتهية غير الحتمية المجمعة وتم توصيلهما بالأتوماتة المنتهية غير الحتمية الأولى والثانية باستخدام الانتقالات الفارغة كما هو موضح في الشكل السابق.

4.1.4 التـكـرار

عند بناء الأتوماتة المنتهية غير الحتمية تقابل التعبير المنتظم*r ووفقاً للفروض السابقة فإنها ستأخذ الشكل التالي :
حيث تم هنا أيضا إضافة حالة بداية وحالة مقبولة جديدان للأتوماتة المنتهية غير الحتمية المدمجة وتوصيلهما مع الأتوماتة المنتهية غير الحتمية الأولى المقابلة للتعبير المنتظم r باستخدام الانتقالات الفارغة وكذلك تم توصيل حالتي البداية في كلاهما مع حالتي القبول كما هو موضح بالشكل لتمثيل عملية التكرار.

والآن وبعد الانتهاء من جميع عمليات التعبيرات المنتظمة و معرفة كيفية تحويلها إلى الأتوماتة المنتهية غير الحتمية يجب الإشارة إلى أن الأتوماتة المنتهية غير الحتمية الناتجة من التحويل يمكن تبسيطها في بعض الأحوال وهذا ما ستوضحه الأمثلة التالية:

مثال 12

لتحويل التعبير المنتظم ab | a إلى الأتوماتة المنتهية غير الحتمية وفقاً لطريقة التحويل السابقة سنبدأ ببناء الأتوماتة المنتهية غير الحتمية المقابلة لكل من التعبيرين المنتظمين a و b واللذان سيأخذان الشكل التالي:
وبالتالي فإن الأتوماتة المنتهية غير الحتمية التي تقابل التعبير المنتظم ab ستكون كما يلي :
لنصل في النهاية للأتوماتة غير الحتمية المقابلة للتعبير المنتظم الأساسي A | ab والذي سيأخذ الشكل التالي:

مثال 13

التعبير المنتظم letter (letter | digit)*i الذي يمثل المعرفات يمكن بناء الـ الأتوماتة المنتهية غير الحتمية المقابل له باستخدام نفس الأسلوب السابق بان نبدأ في البداية إعداد الـ الأتوماتة المنتهية غير الحتمية الذي تقابل كلا من التعبيرين المنتظمين letter , digit كما يلي:

ثم نقوم ببناء الأتوماتة المنتهية غير الحتمية المقابلة للتعبير المنتظم letter | digit كما في الشكل الآتي:

وبعد ذلك نقوم ببناء الأتوماتة غير الحتمية التي تقابل التعبير المنتظم *(letter | digit) كالتالي:

لنصل في النهاية إلى الأتوماتة المنتهية غير الحتمية المقابلة للتعبير المنتظم الأساسي *(letter | digit)ₒletter والتي ستأخذ الشكل الآتي:

تقليل عدد حالات ال DFA

إن الطريقة السابقة الخاصة بتحويل التعبير المنتظم إلى أتوماتة منتهية حتمية قد يعاب عليها أنها تقدم أتوماتة منتهية حتمية معقدة إلى حد ما أكثر من اللازم فمثلا في المثال(10) وصلنا إلى الأتوماتة المنتهية الحتمية التالية التي تقابل التعبير المنتظم *a :

في حين أن الأتوماتة المنتهية الحتمية الآتية يمكن أن تقوم بنفس الدور :

على الرغم من كونها تتكون من عدد حالات أقل وحيث أن كفاءة ال DFA المستخدم في الماسح الخاص بمحلل المفردات يعد من الأمور الهامة لذلك فإنه من المفضل إنشاء الأتوماتة المنتهية الحتمية التي تحتوي على أقل عدد من الحالات وفي الحقيقة وكنتيجة هامة من نظرية الأتوماتة فإن أي أتوماتة منتهية حتمية يقابلها أتوماتة منتهية حتمية وحيدة تتكون من الحد الأدنى من عدد الحالات وهناك أيضا طريقة مباشرة لإيجاد هذه الأتوماتة المنتهية الحتمية عالية الكفاءة من أي أتوماتة منتهية حتمية. وهذه الطريقة تبدأ بأكثر الفروض مثالية حيث يتم إنشاء مجموعتين من الحالات الأولى تتضمن جميع الحالات المقبولة

في الأتوماتة المنتهية الحتمية، بينما تتضمن الثانية باقي الحالات. ثم يتم متابعة جميع الانتقالات الناتجة عن كل حرف من حروف الأبجدية فإذا كان لدى جميع الحالات المقبولة إنتقال نتيجة للحرف a إلى بعض الحالات المقبولة فإنه يتم إنشاء حالة مقبولة موحدة تشمل مجموعة الحالات المقبولة السابقة تتضمن على إنتقال نتيجة للحرف a منها إلى نفسها وبنفس الأسلوب إذا كان لدى جميع الحالات المقبولة نتيجة للحرف a انتقال إلى حالات غير مقبولة فإنه يتم إنشاء حالة مقبولة موحدة تتضمن انتقال نتيجة للحرف a إلى الحالة غير مقبولة المجمعة التي تشمل مجموعة الحالات غير مقبولة السابقة ولكن في نفس الوقت إذا كان هناك لأي حالتين من حالات القبول يوجد إنتقالات نتيجة لنفس الحرف إلى مجموعتين مختلفتين من الحالات أو يوجد لأحدهما إنتقال ولا يوجد لآخر فإنه لا يمكن إنشاء إنتقال موحد لتجميع تلك الحالتين في مجموعة واحدة للحالات المقبولة بل على العكس يجب تقسيم مجموعة الحالات المقبولة إلى عدة مجموعات على حسب حالة الوصول التي يؤدي إليها الإنتقال الناتج عن هذا الحرف ونفس الوضع أيضا ينطبق على أي مجموعة من الحالات فإنه لابد من إعادة الخطوات من البداية ونستمر على ذلك التقسيم لمجموعة حالات الأتوماتة المنتهية الحتمية الأصلية إلى أن نصل إلى وضع لا يمكن معه التقسيم عند ذلك نكون قد وصلنا إلى الأتوماتة المنتهية الحتمية التي تحتوي على الحد الأدنى من عدد الحالات وسوف نوضح أكثر طريقة الوصول إلى تلك الأتوماتة المنتهية الحتمية باستخدام المثالين التاليين:

مثال 18:

بفرض الأتوماتة المنتهية الحتمية التي وصلنا لها في المثال السابق والخاصة بالمفردات فإنها تتكون من أربعة حالات أحدها هي حالة البداية والحالات الثلاثة الباقية هي حالات مقبولة وجميع الحالات المقبولة لديها إنتقالات إلى حالات مقبولة أخرى نتيجة لكل من الحرف letter أو digit ولايوجد اي إنتقال آخر غير ذلك. ولهذا فإنه يمكن تجميع مجموعة الحالات المقبولة في حالة مقبولة واحدة إلى الأتوماتة المنتهية الحتمية التي تحتوي على الحد الأدنى من الحالات والتي تأخذ الشكل التالي:

مثال (19) :

بفرض الأتوماتة المنتهية الحتمية والتي تقابل التعبير المنتظم *ε | a) b)
ففي هذا المثال نجد أن جميع الحالات هي حالات مقبولة وكذلك نلاحظ وجود إنتقال لدى جميع تلك الحالات نتيجة للحرف b يصل إلى حالات مقبولة ممايشير إلى إمكانية تجميع مجموعة الحالات المقبولة في حالة مقبولة موحدة ولكن لكون حرف ال a يفرق بينهم حيث يوجد فقط لدى الحالة (1) إنتقال نتيجة a بينما لايوجد ذلك لدى كل من الحالة (2) والحالة (3) لهذا فإنه لابد من تقسيم المجموعة المقترحة للحالات المقبولة إلى مجموعتين منفصلتين تشمل الأولى الحالة (1) فقط على أن تتضمن الحالتين (2) و (3) في مجموعة واحدة يمثلها حالة مقبولة موحدة في الأتوماتة المنتهية الحتمية المحسنة لتصبح تلك الأتوماتة المنتهية الحتمية التي تحتوي على الحد الأدنى من الحالات كما يلي:
. إنشاء محلل المفردات الخاص باللغة الإفتراضية

نريد في هذا الجزء إعداد البرنامج الخاص بمحلل المفردات الذي يعكس المفاهيم التي تمت دراستها في هذه الوحدة وسيتم ذلك بالنسبة للغة الإفتراضية السابق الإشارة إليها في الوحدة الأولى ونستطيع الآن مناقشة عدد من الموضوعات التطبيقية المتعلقة بهذا الجزء من المترجم.

في الوحدة الاولى تم فقط إعطاء بعض المعلومات البسيطة عن اللغة الإفتراضية ومهمتنا هنا أن نقوم بتحديد الهيكل المفرداتي الخاص بتلك اللغة بشكل كامل عن طريق تعريف مفرداتها والخصائص المرتبطة بها والجدول التالي يلخص مفردات اللغة الإفتراضية مع تصنيفها وفقا للأنواع الأساسية:
حيث تم تقسيم مفردات اللغة إلى ثلاثة مجموعات وهناك ثمانية كلمات محجوزة لها معنى واضح ومتعارف عليه بالإضافة إلى عشرة رموز خاصة تشمل العمليات الحسابية الأربعة الأساسية التي يتم إجراءها على الأعداد الصحيحة وكذلك إثنان فقط من عمليات المقارنة هما علامة التساوي وعلامة الأقل من وأيضاً الأقواس والفاصلة المنقوطة وعلامة التخصيص ويلاحظ أن جميع الرموز الخاصة تتكون من حرف واحد بإستثناء علامة التخصيص التي تتكون من حرفين وباقي المفردات هما الأعداد التي تتكون من سلسلة من واحد أو أكثر من الأرقام العشرية وأيضا المعرفات التي تتكون من أجل التبسيط من سلسلة من حرف واحد أو أكثر من الحروف الأبجدية فقط.

وبالإضافة إلى المفردات تتضمن اللغة التعليقات التي يتم وضعها بين الأقواس ({ }) ويوجد حرية في شكل البرنامج المكتوب بهذه اللغة بإضافة أي من المسافات البيضاء التي تشمل المسافات المفردة والمسافات المتعددة وأيضا رمز السطر الجديد.

ولتصميم ماسح لهذه اللغة فالمفترض أن نبدأ بالتعبيرات المنتظمة ثم بعد ذلك نقوم بإعداد الأتوماتة المنتهية غير الحتمية الأتوماتة المنتهية الحتمية المقابلة لها وفقا للطريقة التي تم شرحها في الجزء السابق وقد تم بالفعل عرض التعبيرات المنتظمة الخاصة بالأعداد و المعرفات وأيضا التعليقات ولكن اللغة الأفتراضية التي نحن بصددها تكون نسخة التعبيرات المنتظمة الخاصة بها أكثر بساطة أما التعبيرات المنتظمة الخاصة بباقي المفردات فإنها في غاية السهولة لأن كلها تتكون من سلسلة حروف ثابتة لهذا فبدلا من المرور بالطريق الخاص بالتحويل وحتى الوصول إلى أتوماتة منتهية حتمية سنقوم بإعداد ال DFA مباشرة، حيث سيتم ذلك في عدة خطوات وسنبدأ بالرموز الخاصة والتي تتكون كلها باستثناء علامة التخصيص من حرف واحد لهذا فإن الأتوماتة المنتهية الحتمية الخاصة بها ستكون كالتالي:

ففي المخطط السابق ستقوم الحالات المقبولة المختلفة بالتفرقة بين المفردات التي سيتم تحديدها بواسطة الماسح وفي حالة استخدام متغير واحد في البرنامج لتحديد المفردة التي تم التعرف عليها فإنه يمكن تجميع الحالات المقبولة المختلفة السابقة في حالة مقبولة واحدة كما يمكن عند ذلك إضافة الجزء الخاص بقبول الأعداد والمعرفات إلى الأتوماتة المنتهية الحتمية لتصبح كما يلي:
ويمكن أيضاً إضافة التعليقات والمسافات البيضاء وعلامة التخصيص للأتوماتة المنتهية الحتمية لتصبح كما يلي:

ولم يتم تضمين الكلمات المحجوزة في ال أتوماتة منتهية حتمية السابقة لأن ذلك هو أسهل أسلوب للتعامل مع تلك الكلمات حيث سيفترض أنها مثل المعرفات وبعد هذا يتم الكشف عنهم في جدول الكلمات المحجوزة بعد قبول هذه المعرفات وهو ما سبق شرحه من قبل عند عرض كيفيه إستخدام جداول الرموز اثناء تحليل المفردات.

وبصفة عامة يحتاج الماسح أيضا إلى تحديد الخصائص الخاصة بالمفردات في حالة وجودها وفي بعض الأحيان أيضا يتم تنفيذ بعض الإجراءات الأخرى مثل إضافة المعرفات إلى جدول الرموز وبالنسبة للغة الإفتراضية فإن الخاصية التي يتم تحديدها هي سلسلة الحروف التي تكون المفردة التي تم التعرف عليها.

تحليـــل الصـــيغ النحويـــة

Syntax Analysis

إن مهمة هذه المرحلة من مراحل المترجم التي تسمى أيضاً بمرحلة الإعراب (Parsing) هي تحديد الصيغ النحوية الخاصة بهيكل البرنامج المراد ترجمته حيث يتم تمثيل الصيغ النحوية الخاصة بلغة البرمجة في شكل قواعد نحوية (Grammar Rules) من القواعد النحوية الخالية السياق (Context free Grammar) بطريقة مشابهة للطريقة المستخدمة في هيكلة المفردات التي يتم تمييزها أثناء مرحلة تحليل المفردات و التي يتم تمثيلها في شكل تعبيرات منتظمة (Regular-free Expression) . وعلى الرغم من أن القواعد النحوية الخالية السياق (التي تم شرحها في الوحدة الثانية) تستخدم تسميات وعمليات مشابهة لمثيلتها في التعبيرات المنتظمة ولكن الفرق الأساسي بينهما في كون القواعد النحوية الخالية السياق تسمح بالقواعد التي تستخدم التكرار (Recursive) كما أوضحنا في الوحدة الثانية من هذه المادة العلمية فعلى سبيل المثال هيكل أمر if يسمح بوجود عدة أوامر if متفرعة منها وهذا التكرار غير مسموح به في التعبيرات المنتظمة وبالإضافة إلى هذا الفرق فإن الخوارزميات المستخدمة في التعرف على تلك الهياكل البرمجية تختلف أيضاً عن خوارزميات تحليل المفردات في كونها تستخدم التكرار وتراكيب البيانات المستخدمة في تمثيل الهياكل النحوية للغة يجب أيضاً أن تكون تكرارية وليست خطية كما في حالة تمثيل المفردات. وعموماً فإن تركيبة البيانات الأساسية في هذه المرحلة هي شكل من أشكال الأشجار تسمى شجرة الإعراب أو شجرة النحو.

. عملية الإعراب

إن تحديد الهيكل النحوي للبرنامج من المفردات التي يتم إنتاجها بواسطة محلل المفردات وما يتبعها من إنشاء شجرة الإعراب أو شجرة النحو التي تمثل ذلك الهيكل هو المهمة الأساسية لمرحلة تحليل الصيغ النحوية أو ما تسمى بمرحلة الإعراب كما ذكرنا سابقاً. ولهذا فإن عملية الإعراب يمكن أن نراها كوظيفة (Function) تأخذ المدخلات المتتابعة من المفردات التي تم توليدها بواسطة محلل المفردات لتقدم كمخرجات شجرة النحو وغالباً لا يتم إعطاء متتابعة المفردات لتلك الوظيفة بشكل مباشر ولكنها تستدعى الوظيفة الخاصة بتحليل المفردات لتطلب المفردة التالية عند الاحتياج إليها وليس قبل ذلك وفي حالة المترجمات التي تتم كمرحلة واحدة (One pass Compilers) والتي سبق شرحها في الوحدة الثالثة فإن مرحلة الإعراب تتضمن جميع المراحل الأخرى للمترجم بما فيها توليد الشفرة المستهدفة من الترجمة ولهذا لا نحتاج لإنشاء شجرة النحو فعلياً وذلك لكون خطوات عملية الإعراب نفسها تمثل شجرة النحو بشكل ضمني كما سبق وأوضحنا ولكن هذا النوع من المترجمات قليل جداً وأغلب المترجمات تتكون من عدة مراحل وفي هذه الحالة فإن شجرة النحو يتم إنشاءها فعلياً لأنها تستخدم كمدخلات للمراحل التالية من المترجم.

ويرتبط هيكل شجرة النحو بشكل كبير بالهيكل النحوي للغة البرمجة وهذه الشجرة دائماً ما يتم تعريفها كتركيبة بيانات متحركة (Dynamic Data Structure) بحيث تتكون كل عقدة داخلها من سجل (Record) يتكون من عدة حقول (Fields) تتضمن عناصر البيانات المطلوبة خلال مراحل الترجمة التالية.

وأحد المشاكل الأكثر صعوبة بالنسبة لمرحلة الإعراب بالمقارنة بمرحلة تحليل المفردات هي كيفية معالجة الأخطاء في حالة محلل المفردات وعند اكتشاف أحد الحروف الذي لا ينتمي لأي من مفردات اللغة فإنه يقوم بإظهار رسالة تفيد وجود خطأ ثم يتخطى هذا الحرف لاستكمال مهمته بينما في حالة حدوث ذلك في مرحلة الإعراب فإن الأمر يتطلب ليس فقط إظهار تقرير بالخطأ ولكن أيضاً لا بد من الاستعفاء من هذا الخطأ حتى يتمكن من متابعة الإعراب ليظهر في النهاية قائمة بجميع الأخطاء التي وجدت. وفي بعض الأحيان يقترح تصحيحاً لهذه الأخطاء أو يقدم برنامجاً خالياً من الأخطاء ولكن هذا

يحدث في الحالات البسيطة والأمر الهام من الناحية التطبيقية بخصوص الاستعفاء من الأخطاء هو أن تكون رسالة الخطأ مفهومة وأقرب ما تكون من الخطأ الفعلي بقدر الإمكان وهذا ليس بالأمر السهل ونظراً لأهمية ذلك فسيتم عرضه بشكل منفصل في الجزء الأخير من هذه الوحدة.
. الإعراب من أعلى إلى أسفل

في طريقة الإعراب من أعلى إلى أسفل (Top down Parsing) يتم إعراب سلسلة المفردات بتتبع الخطوات الخاصة بالاشتقاق من أقصى اليسار وقد سميت هذه الطريقة بهذا الاسم لكونها تقوم بالتنقل داخل شجرة الإعراب أو شجرة النحو بدءاً من جذر الشجرة وحتى أوراقها وذلك على اعتبار أن جذر الشجرة كأحد تراكيب البيانات يكون لأعلى بينما أوراق الشجرة تكون لأسفل.

والإعراب من أعلى لأسفل يتم بأحد شكلين:

1. الإعراب التراجعي (Backtracking Parsing) و

2. الإعراب التنبؤي (Predictive Parsing)

وفي الإعراب التنبؤي يتم التنبؤ بالتركيبة التالية في سلسلة المدخلات عن طريق استخدام واحد أو أكثر من المؤشرات الأمامية (Lookahead) لتتمكن من التعرف على المفردات التالية مقدماً بينما في الإعراب التراجعي يتم تجربة الاحتمالات المختلفة لإعراب المدخلات حيث تتراجع عند الفشل في أي احتمال لتجربة باقي الاحتمالات لحين الوصول إلى الاحتمال السليم وهذا الشكل الأخير من الإعراب يكون فعّالاً ولكنه بطئ ولذلك فهو غير مناسب من الناحية العملية ولن نقوم بدراسته في هذه المادة العلمية وسيقتصر عرضنا فقط على أسلوب الإعراب التنبؤي حيث سندرس طريقتين من طرق الإعراب من أعلى لأسفل كلاهما يستخدم أسلوب الإعراب التنبؤي. الطريقة الأولى تسمى بالإعراب التنازلي التكراري (Recursive Descent Parsing) والثانية تسمى طريقة الإعراب LL (1)i وطريقة الإعراب التنازلي التكراري تعتبر طريقة مناسبة لإنشاء وحدة الإعراب الذي يتم إعدادها يدوياً وسوف يتم عرضه في الجزء القادم على أن يتم دراسة طريقة الإعراب LL (1)i في الجزء الذي يليه وعلى الرغم من صعوبة تطبيق هذه الطريقة الأخيرة من الناحية العملية إلا أنه من المناسب دراستها كنموذج لاستخدام المكدسات (Stacks) في الإعراب كما أنها تساعد في صياغة بعض المشاكل الموجودة في الطريقة الأولى وعموماً فإن تسمية هذه الطريقة بـ LL (1)i لكونها اختصاراً يعنى فيه حرف "L" الأول أن معالجة المدخلات تتم من اليسار (Left) إلى اليمين وذلك لكون بعض طرق الإعراب القديمة كان يتم فيها معالجة المدخلات من اليمن إلى اليسار و حرف "L" الثاني فإنه يشير إلى أنها تقوم بتتبع خطوات الاشتقاق من أقصى اليسار (Left most Derivation) لسلسلة المدخلات أما الرقم "1" الذي بين الأقواس فإنه يعنى أنه يستخدم مفردة واحدة من المدخلات للتنبؤ باتجاه الإعراب السليم وهناك طرق إعراب تنبؤية أخرى تستخدم عدة مفردات للتنبؤ باتجاه الإعراب السليم.
. الإعراب التنازلي التكراري

إن الفكرة الخاصة بالإعراب التنازلي التكراري فكرة بسيطة كونه يرى القاعدة النحوية للرمز اللانهائي A كتعريف للإجراء الذي سيتعرف على A حيث يحدد الطرف الأيمن لتلك القاعدة النحوية هيكل الخطوات الخاصة بهذا الإجراء فالرموز النهائية يقابلها تطابق مع المدخلات والرموز اللانهائية يقابلها استدعاء لإجراءات أخرى. أما البدائل فيقابلها أحد أوامر الاختيار مثل if أو case ولتوضيح ذلك بفرض القواعد النحوية التالية الخاصة بالتعبيرات (Expressions) الرياضية:

<exp> <exp> <addop> <term> I <term>

- | + <addop>

<term> <term> <mulop> <factor> I <factor>

* <mulop>

factor> (<exp>) | number>

فإن الإجراء التنازلي التكراري الذي يتعرف على الرمز اللانهائي <factor> يمكن أن يكون كما يلي:

```
procedure factor
begin
case lookahead of
('(') match ('('); exp ; match : ')'
('number' : match ('number'
else error
end case
end factor
```

وفي هذا الإجراء السابق نفترض أن المتغير lookahead يحتفظ بالمفردة القادمة في المدخلات وأن الإجراء match يقوم بمطابقة تلك المفردة مع المعطيات على أن يتم تقديم مؤشر المدخلات إذا حدث ذلك التطابق أو يظهر رسالة خطأ إذا لم يحدث ذلك وبالتالي فإن هيكل الإجراء match سيكون كما يلي:

```
( procedure match (t : token
begin
if lookahead = t then
```

lookahead: = next token

else

error

end match

وبالطبع عند استدعاء الإجراء match مع كل من المفردة ')' والمفردة 'number' لا بد أن يحدث التطابق بين lookahead وبين t ولكن ذلك ليس بضروري حدوثه في حالة استدعائه مع المفردة '(' ولهذا كان من الضروري اختبار ذلك داخل الإجراء match .

وبالرغم من سهولة إعداد الإجراء التنازلي التكراري للرمز اللانهائي factor كما شاهدنا، ألا أن الأمر ليس كذلك بالنسبة لباقي القواعد النحوية الخاصة بالتعبير الرياضي. ويستلزم الأمر إدخال بعض التغييرات عليها حتى يمكنا إعداد الإجراء التنازلي التكراري الذي يمثلها. وهذا ما سيتم ولكن قبل هذا نفترض كمثال آخر القاعدة النحوية المبسطة لأمر if :

<if ▯ stmt > if (<exp>) <stmt >

<if (<exp>) <stmt> else <stmt |

والتي يمكن إعداد الإجراء التنازلي التكراري الخاص بها ليكون كما يلي:

procedure ifstmt

begin

; ('match ('if

; (')') match

; exp

; (')') match

; stmt

if lookahead = 'else' then

; ('match ('else

; stmt

end if

end ifstmt

ففي هذا المثال لا نستطيع أن نفرق مباشرة بين الاختيارين الموجودين في الجانب الأيمن من القاعدة النحوية لأن كلاهما يبدآن بالرمز النهائي 'if' وبدلاً من ذلك فإنه يجب وضع الاختبار الذي يفرق بينهما عند إدراك الجزء الاختياري else بظهور المفردة 'else' في المدخلات.

1.3 استخدام شكل EBNF في تمثيل الاختيار والتكرار

في المثال السابق الخاص بأمر if نجد أن الإجراء التنازلي التكراري قد قام فعلياً بتمثيل القاعدة النحوية التالية التي تستخدم شكل باكوس نور الموسع :

[<if ⍰ stmt > if (<exp>) <stmt> [else <stmt >

وليس الشكل الأصلي لباكوس نور حيث تم استخدام الأقواس [ ] الخاصة بشكل باكوس نور الموسع للتعبير عن الجزء الاختياري في أمر if وهو ما تم تمثيله في الإجراء التنازلي التكراري من خلال الاختبار. وفي الحقيقة فإن شكل باكوس نور الموسع قد تم تصميمه ليعكس بقدر الإمكان الخطوات الفعلية للإعراب التنازلي التكراري. ولهذا فإن القواعد النحوية لا بد من إعدادها بواسطة هذا الشكل في حالة استخدام الإعراب التنازلي التكراري. ويجب أيضاً ملاحظة أن القاعدة النحوية السابقة تحتوي على بعض الغموض كما أوضحنا ذلك في الوحدة الثانية لهذا فإنه من الطبيعي إعداد وحدة الإعراب التي تقوم بمطابقة المفردة 'else' بأسرع ما يمكن في كل مرة تظهر فيها في المدخلات والذي تتمشى مع القاعدة غير الغامضة الخاصة بحالة التداخل في أمر if .

والآن نعود إلى القواعد النحوية الخاصة بالتعبير الرياضي المبسط وبفرض القاعدة النحوية التالية الخاصة بالرمز اللانهائي <exp> والمكتوبة باستخدام شكل باكوس نور:

<exp> <exp> <addop> <term> │ <term>

فإذا حاولنا إعداد الإجراء التنازلي التكراري الخاص بهذا الرمز اللانهائي <exp> وفقاً للتمثيل المباشر للقاعدة النحوية السابقة فإن أول خطوة في هذا الإجراء ستكون استدعاء الإجراء لنفسه وهو ما يؤدي مباشرة لحدوث تكرار غير منتهٍ وهي المشكلة التي سبق الإشارة إليها في الوحدة الثالثة والخاصة بالتكرار من ناحية اليسار ولكننا الآن ولحل هذه المشكلة وبدلاً من تحويل التكرار ليكون من ناحية

اليمين كما فعلنا سابقاً، فإنه يمكن إعادة كتابة القاعدة السابقة باستخدام شكل باكوس نور الموسع والتي ستحل تلك المشكلة وستصبح كما يلي:

{ <exp> <term> { <addop> <term>

مع الأخذ في الاعتبار بأن الأقواس { } في شكل باكوس نور الموسع يتم تمثيلها من خلال أحد الأوامر التكرارية مثل أمر while وهذا ما يتضح في الإجراء التنازلي التكراري التالي الخاص بالرمز <exp> اللانهائي:

```
procedure exp
begin
;term
while lookahead = '+' or lookahead = '-' do
; (match (lookahead
term
end while
end exp
```

وبنفس الطريقة فإن القاعدة النحوية الخاصة بالرمز اللانهائي < term> عند إعادة كتابتها باستخدام شكل باكوس نور الموسع ستكون كما يلي:

{ <term> < factor > { <mulop> < factor>

وبالتالي سيكون الإجراء التنازلي التكراري الذي يقابلها كالآتي:

```
procedure term
begin
; factor
```

while lookahead = '*' do

; (match (lookahead

factor

end while

end term

وقد تم في الإجراءين الخاصين بالرمزين <exp> و <term> اللانهائيين تجاوز أعداد إجراء مستقل للرمزين <addop> و <mulop> حيث أنهما مجرد خطوة تطابق مع العمليات الرياضية الأساسية وتم الاكتفاء بالقيام بهذا التطابق مباشرة كما رأينا في الإجراءين السابقين.

والسؤال الذي يطرح نفسه الآن هل الاتحاد من ناحية اليسار والذي يظهر واضحاً في القاعدة النحوية المكتوبة باستخدام شكل باكوس نور ما زال موجوداً في مثيلتها المكتوبة باستخدام شكل باكوس نور الموسع؟ فإذا فرض على سبيل المثال أننا نريد إعداد آلة حاسبة باستخدام طريقة الإعراب التنازلي التكراري تطبق القواعد النحوية الخاصة بالتعبير الرياضي بشكله المبسط الذي تم عرضها في هذا الجزء ونستطيع التأكد من اتحاد العمليات من ناحية اليسار عن طريق تنفيذ تلك العمليات داخل أمر التكرار وهو ما يتضح بفرض تحويل الإجراءات الخاصة بالإعراب التنازلي التكراري إلى وظائف برمجية ترجع أعداداً صحيحة فمثلاً الوظيفة التي تمثل الرمز اللانهائي <exp> ستكون كما يلي:

function exp: integer

var temp: integer

begin

; temp := term

while lookahead = '+' or lookahead = '-' do

case lookahead of

; ("+') match : '+'

; temp := temp + term

; ('-') match :'-'

; temp := term ? term

end case

end while

end exp

وبنفس الطريقة يتم إعداد الوظائف الخاصة بباقي الرموز اللانهائية وقد تم استخدام تلك الفكرة في إنشاء آلة حاسبة بسيطة بلغة السي والبرنامج الكامل التالي يقدم تلك الآلة الحاسبة:

```
<include <stdio.h #
<include <stdbb.h #
/* char token ; /* global variable
/* function prototypes far recursive calls */
; (int exp (void
; (int term (void
; (int factor (void
(void error (void
}
; ("fprintf (stderr , "Error\n
; (exit (1
{
(void match ( char expectedToken
}
( if ( token = = expectedToken
```

```
; token = getchar
else error
{
main
}
; int result
; token = getchar
; result = exp
( 'if ( token = = '\n
;(printf ( "Result = %d \n", result
; else error
;return 0
{
( int exp ( void
}
; int temp = term
(('-' = = while (( token = = '+') || (token
} ( switch ( token
; ('+' ) case '+' : match
; temp + = term
; break
```

```
; ('-' ) case '-' : match
;  temp - = term
;break
{
; return temp
{
( int term ( void
}
;  int temp = factor
} ('*' = = while ( token
} ('*' ) match
;  temp * = factor
{
; return temp
{
( int factor ( void
}
; int temp
} (')' = = if ( token
; (')') match
;  temp = exp
```

```
; ('(') match
{
} (( else if ( isdigit ( token
; ( ungetc ( token , stdin
; ( scanf ("%d" , &temp
;  token = getchar
{
;  else error
; return temp
{
```

إن هذه الطريقة لتحويل القواعد النحوية المكتوبة باستخدام شكل باكوس نور الموسع إلى إجراءات أو وظائف برمجية ذات كفاءة عالية ولكنها تتطلب قدراً كبيراً من الاهتمام بترتيب خطوات كل إجراء أو وظيفة وفق قاعدة عامة لا بد أن يقف المؤشر الذي يمثله المتغير lookahead عند أول مفردة قبل البدء في عملية الإعراب ولا يتم طلب مفردة جديدة إلا بعد اختبار المفردة الحالية وهو ما يتم داخل الإجراء match في العرض السابق وهذا الاهتمام بجدولة الأحداث يحدث أيضاً أثناء بناء شجرة النحو حيث لا بد من بناء الشجرة والعقد التي تتكون منها بترتيب وتسلسل محدد وفقاً لترتيب ظهور المفردات في المدخلات.

وعموماً فإن أسلوب الإعراب التنازلي التكراري الذي عرضناه في هذا الجزء يتميز بالسهولة والمرونة مما يسمح للمبرمج من ضبط الجدولة السليمة للأحداث وذلك ما يجعل تلك الأسلوب الاختيار المناسب عند إعداد محلل الصيغ النحوية يدوياً وهو ما يحدث في حالة لغات البرمجة صغيرة الحجم مثل اللغة الافتراضية الذي تم تقديمها في الوحدة الأولى. ولكن في حالة لغات البرمجة الحقيقية والكبيرة الحجم مثل لغة السي أو غيرها من اللغات المعروفة الأخرى فإن القيام بذلك يكون في غاية الصعوبة ودائماً ما يتم اللجوء إلى توليد محلل الصيغ النحوية أوتوماتيكياً. وهناك عدة أسباب لذلك. الأول: هو صعوبة تحويل القواعد النحوية من شكل باكوس نور إلى شكل باكوس نور الموسع. وسندرس في الجزء القادم بديلاً لاستخدام الشكل الموسع ولكنه مكافئ له والسبب الثاني: عند التعامل مع قاعدة نحوية تحتوي على بدائل مثل القاعدة التالية:

. . . | A> | β>

قد تكون هناك صعوبة في تحديد أي البدائل يتم استـــخدامها ومتـــى يتم اختـــيارµ <A> ومتى يتم اختيار β <A>i إذا كانت كل من و β يبدآن برمز لا نهائي. ولاتخاذ القرار المناسب فإن الأمر يتطلب تحديد مجموعة المفردات التي يمكن أن تبدأ بها كل من و β وهي ما تسمى مجموعة البداية أو المجموعة الأولى (First Set) وهذه المجموعة سبق الإشارة إليها في الوحدة الثالثة وسنتعرض لها ببعض التفصيل أيضاً في الجزء بعد القادم. أما السبب الثالث فهو عند كتابة معالجة قاعدة نحوية كالتالية:

A> Є>

فإن الأمر يستلزم معرفة أي المفردات تأتي بعد الرمز اللانهائي <A> حيث أن تلك المفردات تعنى أن هذا الرمز قد لا يظهر فعلياً عند ذلك الموضع في الإعراب ومجموعة المفردات هذه تسمى المجموعة المتتالية (Follow Set) وسوف نتعرض لها بالتفصيل في الجزء بعد القادم أيضا.

وهناك اهتمام وأهمية كبيرة لتجهيز كل من المجموعة الأولى والمجموعة المتتالية ليس فقط لمعالجة حالات القواعد النحوية التي أشرنا إليها هنا ولكن أيضاً لأنها تساعد في اكتشاف الأخطاء في المدخلات مبكراً مع إمكانية الاستعفاء عنها وهو ما سنوضحه في الجزء الأخير من هذه الوحدة.

. الإعراب LL (1)i

هذا الأسلوب يقوم بعملية الإعراب باستخدام المكدس (Stack) بشكل مباشر بدلاً من الاستدعاء التكراري الذي يستخدم في الأسلوب السابق للإعراب وتمثيل هذا المكدس بالطريقة المعتادة يساعد على انجاز عملية الإعراب بسهولة وبسرعة عالية ولتوضيح ذلك الأسلوب نستخدم القاعدة النحوية البسيطة التالية التي تولد سلسلة من الأقواس المتوازنة:

S> (<S>) <S> | Є>

وبفرض سلسلة المدخلات فإن الجدول التالي يوضح خطوات عملية الإعراب من أعلى إلى أسفل لتلك المدخلات :

حيث يتكون الجدول من ثلاثة أعمدة بالإضافة إلى الأرقام التي على اليسار والتي توضح ترتيب الخطوات، بينما يبين العمود الأول من الجدول محتويات مكدس الإعراب وقاع المكدس يكون ناحية اليسار. أما قمته فناحية اليمين ونرمز لقاع المكدس بعلامة الدولار ($) وبالتالي فإن المكدس عندما يحتوي على الرمز اللانهائي S في قمته فإنه سيظهر كما يلي:

S $

على أن يتم إضافة العناصر التالية من ناحية اليمين. أما العمود الثاني من الجدول فيظهر حروف المدخلات الذي يتم سردها من اليسار إلى اليمين مع استخدام علامة الدولار ($) للإشارة إلى نهاية المدخلات في حين يعطى العمود الثالث والأخير من الجدول وصف مختصر للإجراء الذي يتم اتخاذه لتغيير محتويات المكدس والمدخلات والذي تظهر نتيجته في الصف التالي من الجدول.

هذا وتبدأ عملية الإعـراب من أعلى إلى أسفل بإضـافة رمـز البدايـة (Start Symbol) الخاص بالقواعد النحوية التي تستخدم في تحليل الصيغ النحوية إلى مكدس الإعراب ويتم قبول سلسلة المدخلات بعد تتابع الإجراءات إذا أصبح كل من المكدس والمدخلات فارغين أي أن النظام العام للعملية الناجحة للإعراب من أعلى إلى أسفل يكون كالآتي:

$ Start Symbol Input String $

قبول $ $

وفي حالة المثال السابق فإن رمز البداية هو S وسلسلة المدخلات هي ويتم الإعراب باستبدال الرمز اللانهائي الذي يظهر في قمة المكدس بأحد الاختيارات من القاعدة أو القواعد النحوية الخاصة بهذا الرمز والمكتوبة باستخدام شكل باكوس نور حيث يتم الاختيار الذي يجعل مفردة المدخلات الحالية تظهر في قمة مكدس الإعراب وبالتالي يحدث التطابق مع مفردة المدخلات وعند ذلك يتم حذف المفردة المتطابقة من كل من المكدس والمدخلات أي أن الإجراءين الأساسيين للإعراب من أعلى لأسفل هما:

1- استبدل الرمز اللانهائي A الذي يظهر على قمة المكدس بالسلسلة عن طريق استخدام اختيار القاعدة النحوية .

2- مطابقة المفردة التي على قمة المكدس مع مفردة المدخلات .

ويجب ملاحظة أن السلسلة البديلة للرمز اللانهائي A يجب إضافتها إلى مكدس الإعراب بترتيب عكسي أي من اليمين إلى اليسار وذلك لكي نضمن ظهور بداية السلسلة على قمة المكدس فمثلاً في الخطوة الأولى من الجدول السابق فإن المكدس والمدخلات يكونان كما يلي :

$ S $

والقاعدة النحوية التي تستخدم لاستبدال الرمز اللانهائي S الذي على قمة المكدس هي

s (s) s بحيث يتم إضافة السلسلة)S(S إلى المكدس ليصبح المكدس والمدخلات كما يلي:

$ S) S($

والآن يظهر القوس الأيسر على قمة المكدس الذي يتطابق مع مفردة المدخلات لكي نصل إلى الوضع التالي:

$ S) S)$

. التجميع اليساري وحذف التكرار اليساري

يعاني التكرار والاختيار في طريقة الإعراب LL(1)i من نفس المشاكل التي تحدث لهما في طرقة الإعراب التنازلي التكراري ولهذا السبب لم نستطع حتى الآن إعطاء جدول الإعراب للقواعد النحوية الخاصة بالتعبير الرياضي البسيط الذي سبق تقديمه في الجزء السابق وبينما كان الحل لتلك المشكلة في طريقة الإعراب التنازلي التكراري هو استخدام الشكل EBNF لكننا لا نستطيع تطبيق نفس الفكرة في طريقة الإعراب LL(1)i وبدلاً من ذلك يجب أن نقوم بإعادة كتابة القواعد النحوية باستخدام الشكل BNF المعتاد ولكن بصورة يمكن أن يقبلها الخوارزمية الخاصة بطريقة الإعراب LL(1)i والأسلوبين المعتادين الذين يمكن أن نطبقها في هذه الحالة هما حذف التكرار اليساري (Left Recursion Removal) والتجميع اليساري (Left Factoring) وهذين الأسلوبين سيتم شرحهما في هذا الجزء مع الأخذ في الاعتبار ان تطبيقهما ليس من الضروري أن يضمن أن تحويل القواعد النحوية إلى قواعد نحوية مقبولة بواسطة الإعراب LL(1)i وذلك مثل ما كان بالنسبة للشكل EBNF الذي لا يضمن أيضاً حل جميع المشاكل عند أعداد الإعراب التنازلي التكراري ولكن مع ذلك فإن هذين الأسلوبين يكونان فعالان جداً في أغلب الحالات التطبيقية ولهما ميزة إمكانية تطبيقهما أوتوماتيكيا وبافتراض نجاح ذلك فإن الإعراب LL(1)i باستخدام تلكما الأسلوبين يمكن أن يتم توليده أوتوماتيكيا.

حذف التكرار اليساري

إن التكرار اليساري (Left Recursive) شائع الاستخدام لجعل اتحاد العمليات من ناحية اليسار كما في القاعدة النحوية التالية الخاصة بالتعبير الرياضي البسيط:

exp exp addop term | term

التي تجعل العمليات التي يمثلها addop ذات اتحاد من ناحية اليسار وهذه الحالة من التكرار اليساري تعد من أبسط الحالات وهي حالة وجود اختيار وحيد في القاعدة النحوية يحتوي على تكرار يساري ولكن الأمر يصبح أكثر تعقيداً في حالة وجود أكثر من اختيار في القاعدة النحوية يحتوي على مثل هذا التكرار اليساري وهو ما يحدث في حالة سرد عمليات addop كما في القاعدة النحوية التالية :

exp exp + term | exp term | term

وعموماً فإن كلا الحالتين يتضمنان على تكرار يساري مباشر حيث أن التكرار اليساري يحدث فقط خلال نفس القاعدة النحوية الخاصة برمز لانهائي واحدة مثل exp في القاعدة النحوية السابقة والحالة الاكثر صعوبة هي حالة التكرار اليساري غير المباشر كما في القاعدتين النحويتين التاليتين:

وهذه الحالة الأخيرة غالباً ما لا تحدث في القواعد النحوية الخاصة بلغات البرمجة الفعلية ولكننا سنقدم حل لتلك الحالة أيضاً من أجل استكمال الحالات الثلاثة.

الحالة الأولى : حالة التكرار اليساري المباشر البسيط

في هذه الحالة يكون التكرار اليساري في القاعدة النحوية على الشكل:

A A α | B

حيث كلاً من α و يمثلان سلسلة من الرموز النهائية واللانهائية ولا تبدأ بالرمز اللانهائي A ولحذف التكرار اليساري فانه يعاد كتابة هذه القاعدة النحوية في قاعدتين الأولى تقوم بتوليد أولاً والأخرى تولد التكرار الخاص بـ α باستخدام التكرار من ناحية اليمين بدلاً من التكرار اليساري كما يلي :

'A→ B A

A→ α A' | Є

مثال 1

بفرض أحد القواعد النحوية الخاصة بالتعبير الرياضي البسيط السابق الإشارة إليها :

exp → exp addop term | term

فهذه القاعدة على شكل الحالة الأولى حيث A = exp , α = addop term , B = term وبالتالي سيعاد كتابتها لحذف التكرار اليساري لتصبح كما يلي :

'Exp → term exp

Exp' → addop term exp' | Є

الحالة الثانية : الحالة العامة للتكرار اليساري المباشر

في هذه الحالة تكون القاعدة النحوية على الشكل الآتي:

حيث لا يبدأ أي من بالرمز اللانهائي A ويكون الحل في تلك الحالة مشابه للحالة البسيطة السابقة مع زيادة عدد الاختيارات كما يلي :

مثال 2

بفرض القاعدة النحوية التالية :

exp exp + term │ exp term │ term

فسنقوم بحذف التكرار اليساري كما يلي :

'exp term exp

exp' + term exp' │ - term exp' │ Є

الحالة الثالثة : حالة التكرار اليساري العامة

هذه الحالة العامة تغطي جميع احتمالات التكرار اليساري سواء البسيط أو غير البسيط وأيضاً المباشر أو الغير مباشر وعند ذلك نقوم باستخدام خوارزمية تضمن العمل بنجاح في حالة القواعد النحوية التي لا تحتوي على أي قاعدة نحوية ذات سلسلة فارغة وكذلك لا تتضمن أي دائرة (cycle) ويقصد بالدائرة إمكانية وجود اشتقاق من خطوة واحدة على الأقل يبدأ وينتهي بنفس الرمز اللانهائي أي كما يلي :

A α *A

لأن وجود مثل هذه الدائرة في القواعد النحوية يجعل خوارزمية الإعراب تدخل في حالة تكرار غير منتهية (Infinite loop) وبصفة عامة لا تتضمن القواعد النحوية الخاصة بأي لغة من لغات البرمجة على مثل هذه الدائرة كما أنها لا تحتوي على قواعد نحوية ذات سلسلة فارغة إلا في حدود ضيقة جداً وبشكل مقيد لذلك فإن الخوارزمية الذي نقدمه هنا سوف يعمل دائماً لمثل هذه القواعد النحوية حيث تقوم هذه الخوارزمية بترتيب الرموز اللانهائية التي تشملها اللغة بأي ترتيب وليكن AN A1 A2 وبعد ذلك يتحاشى وجود أي قاعدة نحوية على الشكل : Ai Aj Y

عندما تكون i _> J وإذا تم ذلك لجميع قيم الـ i فإن هذا يعني عدم ترك أي تكرار لان الفهرس في هذه الحالة يكون تصاعدياً ونفس الفهرس لا يمكن الوصول إليه مرة أخرى وفيما يلي خطوات تلك الخوارزمية :

For i := 1 to m do

For j := 1 to i - 1 do

replace each grammar rule choice of

the farm Ai Aj B by the same rule

Ai α1 B | α2B | . . . | αk B where

Aj α1 | α2 | . . . | αK in the current rule

For Aj

مثال 3

بفرض القواعد النحوية التالية :

A Ba | Aa | c

B Bb | Ab | d

وبالطبع أن تلك القواعد افتراضية بالكامل لا يمكن حدوثها في أي لغة من لغات البرمجة المعتادة وفي هذا المثال نفترض أن الرمز اللانهائي B رقمه أعلى من الرقم الخاص بالرمز اللانهائي A (بمعنى أن A1=A و A2=B) حتى يمكن تنفيذ الخوارزمية السابقة وحيث أن n=2 فإن أمر for الخارجي في الخوارزمية يتم تنفيذه مرتان وعندما يكون i=1 فإن أمر for الداخلي لن يتم تنفيذه وبالتالي فإن الإجراء الوحيد الذي سيتم هو حذف التكرار اليساري المباشر للرمز اللانهائي A لتصبح القواعد النحوية السابقة كما يلي:

'A B a A' | c A

A' a A' | Є

B Bb | A b | d

أما عندما يكون i=2 فإن أمر for الداخلية سيتم تنفيذه مرة واحدة ويكون فيه j=1 وفي هذه الحالة سيتم حذف القاعدة النحوية B Ab عن طريق استبدال الرمز اللانهائي A بكل اختياراته من القاعدة الأولى وعند ذلك تصبح القواعد النحوية كا لآتي :

'A B a A' | cA

A' aA' | Є

B Bb | Bb | Ba A'b | cA'b | d

وفي النهاية يتم حذف التكرار اليساري المباشر للرمز اللانهائي B ونصل إلى القواعد النحوية التالية التي لا تحتوي على أي تكرار يساري:

'A BaA' | cA

A' aA' | Є

'B cA'bB' | d B

B' b B' | a A' b B' | Є

الآن وبعد الانتهاء من حل جميع حالات مشكلة التكرار اليساري يجب التأكيد على أن حذف التكرار اليساري من القواعد النحوية الخاصة بأي لغة من اللغات لا يؤثر على اللغة نفسها ولكنه يحدث بعض التغير في تلك القواعد النحوية وأيضاً في شجرة الإعراب. وهذا التغير يؤدي إلى تعقيد عملية الإعراب وتصميم وحدة الإعراب الخاصة بهذه اللغة ولتوضيح ذلك نعود إلى القواعد النحوية الخاصة بالتعبير الرياضي البسيط التي يتم استخدامها كمثال والتي تتضمن عدة تكرارات يسارية وذلك للتعبير عن الاتحاد اليساري للعمليات فإذا تم حذف هذه التكرارات اليسارية المباشرة من تلك القواعد النحوية ستصبح كما يلي:

'exp term exp

exp' addop term exp' | Є

- | + addop

'term factor term

term' mulop factor term' | Є

* mulop

factor (exp) | number

وبفرض التعبير الرياضي 5-4-3 فإن شجرة الإعراب التي تمثل ذلك التعبير ستكون كالآتي :

حيث يلاحظ أن تلك الشجرة لا توضح الاتحاد اليساري لعملية الطرح بل على العكس تظهر أن الاتحاد الخاص بهذه العملية من ناحية اليمين وذلك كنتيجة لحذف التكرار اليسار واستبداله بتكرار يميني كما هو واضح في القواعد النحوية من خلال الإعراب لابد أن ينشأ شجرة النحو

السليمة التي تعكس الاتحاد اليساري لعملية الطرح والتي تكون كما يلي :

وهذه المهمة ليست بالبسيطة عند استخدام القواعد النحوية الأخيرة ولكن دعنا نوضح كيفية القيام بذلك في حالة شجرة الإعراب السابقة ولكي يتم هذا فإن القيمة 3 يجب تمريرها من جذر الشجرة exp إلى ابنهما الأيمن التي تمثله العقدة 'exp حيث يتم طرح القيمة 4 على أن يمرر الناتج 1- إلى ابنها الذي على أقصى اليمين العقدة 'exp الثانية التي تقوم بدورها بطرح 5 لتصل إلى الناتج 6- الذي يتم تمريره إلى العقدة 'exp الأخيرة التي لا يوجد لها إلا ابناً واحداً هو السلسلة الفارغة Є لذلك فهي تقوم بإرجاع التابع الأخير إلى أعلى شجرة الإعراب ليصل في النهاية إلى جذر الشجرة الممثل بالعقدة exp ويكون ذلك هو القيمة النهائية للتعبير الرياضي 5-4-3 التعبير الرياضي عن تطبيق الاتحاد اليساري لعملية الطرح. ونوضح الآن كيفية القيام بذلك العمل في أسلوب الإعراب التنازلي التكراري السابق عرضه في الجزء السابق وبالنسبة للقواعد النحوية الأخيرة الخاصة بالتعبير الرياضي البسيط بعد حذف التكرار اليساري منها سيكون كل من الإجراءين exp و 'exp كما يلي:

```
procedure exp
begin
; term
; 'exp
; end exp
' procedure exp
begin
case lookahead of
; ('+') match ; '+'
; term
; 'exp
; ('-') match : '-'
; term
```

; 'exp

; end case

; 'end exp

ولجعل الإجراءين السابقين يقومان بحساب قيمة التعبير الرياضي بشكل فعلي نحتاج إلى إعادة كتابتهما ليصبحا كما يلي:

function exp ; integer

; var temp ; integer

begin

; temp : = term

; (return exp' (temp

; end exp

function exp' (valuesofor: integer): integer

begin

if lookahead = '+' or lookahead = '?then

case lookahead of

; ('+') match : '+'

; valuesofor : = valuesofor + term

; ('-') match : '-'

; valuesofor : = valuesofor ? term

; end case

; (return exp' (valuesofor

; else return valuesofor

; 'end exp

ويلاحظ أن الإجراء exp' احتاج في وضعه الأخير أن تمرر له معاملات من الإجراء exp وهو موقف مشابه لما يحدث من تلك الإجراءات عند إرجاع شجرة النحو التي تعكس الاتحاد اليساري للعمليات.

وفي النهاية يجب ملاحظة أن القواعد النحوية في وضعها الأخير بعد حذف التكرار اليساري هي قواعد نحوية في صالحة لتطبيق طريقة الإعراب LL(1)i عليها وذلك باستخدام جدول الإعراب التالي الذي سنشرح كيفية إعداده في الجزء القادم:
التجميع اليساري

يكون التجميع اليساري مطلوب عند وجود أكثر من اختيار في القاعدة النحوية يشتركان في جزء من أقصى يسار الاختيار كما في القاعدة التالية

A α B | α Y

وكمثال آخر من لغات البرمجة القاعدة النحوية التالية الخاصة بأمر if :

If stmt if (exp) statement

| if (exp) statement else statement

وتكمن المشكلة بالنسبة لطريقة الإعراب LL(1)i في عدم القدرة على التمييز بين الاختيارات في هذه الحالة ويكون الحل في تجميع الجزء المشترك بين الاختيارات مع اعادة كتابة القاعدة النحوية والقاعدة الأولى من هذا الجزء تصبح كما يلي بعد التجميع وإعادة كتابتها :

'A α A

A' B | Y

أو أن تكتب كالآتي مع استخدام الأقواس:

(A α (B | Y

وهو الشكل الذي يشبه كثيراً عملية التجميع في التعبيرات الرياضية وعملية التجميع من ناحية اليسار يمكن أن تتم في حالة وجود أكثر من اختيارين في القاعدة النحوية والخوارزمية التالية تقدم الحالة العامة لاجراء التجميع اليساري:

While these are changes to the grammar do

for each nonterminal A do

let be a prefix of maximal length that is shared by two or more -

production choices for A

if α = Є then

let A α1 | α2 | . . . | αn be all the Production choices for -

: A and sppose that α1 , . . . , αk share α so that

and the share no common prefix

αn do not share α, ,and the αk+1

:replace the rule A α1 | α2 | . . . | αn By the rules -

مثال 4

بفرض القاعدة النحوية التالية الخاصة بتتابع الأوامر في لغات البرمجة والتي تحتوي على تكرار من ناحية اليمين :

stmt ⍰ sequence stmt; stmt ⍰ sequence | stmt

حيث يلاحظ وجود جزء مشترك بين الاختيارين من ناحية اليسار وبالتالي يمكن إجراء التجميع مع إعادة كتابة القاعدة النحوية لتصبح كما يلي :

'stmt ⍰ sequence stmt stmt ⍰ seq

stmt ⍰ seq' ; stmt ⍰ sequence | Є

مثال 5

بفرض القاعدة النحوية التالية الخاصة بأمر if :

If ⍰ stmt if (exp) statement

if (exp) statement else statement │

فإن التجميع اليساري لهذه القاعدة النحوية سيجعلها تصبح كالآتي :

If stmt if (exp) statement else post

Else - post else statement │ Є

مثال 6

بفرض كتابة القاعدة النحوية الخاصة بالتعبير الرياضي ولكن لاحد العمليات ذات الاتحاد من ناحية اليمين وليكن الاس فانها ستكون كما يلي :

exp term * * exp │ term

فإن هذه القاعدة النحوية تحتاج إلى تجميع يساري لتصبح كما يلي :

'exp term exp

exp' * * exp │ Є

وفي النهاية يجب ملاحظة أن جميع القواعد النحوية التي في الامثلة السابقة وبعد تطبيق التجميع اليساري عليهم أصبحت صالحة لاستخدام أسلوب الاعراب LL(1)i معها وسوف نقوم بإعداد الاعراب الخاص ببعضهم في الجزء القادم .

3.4 بناء شجرة النحو في الاعراب LL(1)i

إن الإعراب LL(1)i قد يقوم ببناء شجرة النحو بدلاً من شجرة الاعراب التي تم استخدامها قبل ذلك عند تطبيق هذا الاسلوب من الاعراب وقد سبق وشاهدنا تطبيق أسلوب الإعراب التنازلي التكراري إن استخدام شجرة النحو سهل نسبياً ولكن تعديل اسلوب الإعراب LL(1)i لكي يقوم باستخدام شجرة النحو أكثر صعوبة وذلك لكون هيكل شجرة النحو قد يتم تغييره كلياً نتيجة للتجميع اليساري أو حذف التكرار اليساري وهذا نتيجة لأن مكدس الإعراب يمثل فقط التراكيب المتوقعة وليس التراكيب الفعلية ولذلك يتم تأخير بناء عقد شجرة النحو لحين حذف التراكيب من مكدس الإعراب وليس عند إضافتها له وعموماً فإن هذا يتطلب إستخدام مكدس إضافي لمتابعة عقد شجرة النحو على أن يقوم مكدس الإعراب بتحديد ما يتم إجراءه على هذا المكدس الجديد وتوقيت حدوثه.

5. المجموعات الأولى والمجموعات المتتالية

لتطبيق الخوارزمية الخاصة بأسلوب الإعراب LL(1)i فإن الأمر يستلزم تطوير خوارزمية لبناء جدول الإعراب المستخدم مع ذلك الأسلوب وهذا يتطلب كما أشرنا لذلك من قبل حساب كل من

المجموعات الأولى (First Sets) والمجموعات المتتالية (Follow Sets) وسوف نقوم في هذا الجزء بتعريف وتجهيز هذه المجموعات وبعد ذلك نقوم بتقديم وصف دقيق لكيفية انشاء الإعراب LL(1)i .

1.5 المجموعات الأولى

إذا كانت X هي أحد رموز القواعد النحوية سواء كانت نهائية أو لا نهائية أو حتى السلسلة الفارغة Є فإن First (X)i تتكون من رموز نهائية و في بعض الحالات وفقا للقواعد الآتية :

1- إذا كانت x هي رمز نهائي أو Є فإن {x} =ه first(x)i .

2- إذا كانت x هي رمز نهائي فلكل اختيار في القواعد النحوية على الشكل x → x1x2xn فإن first(x)i تحتوي على { } - first(x1)i وإذا كانت أيضا كل المجموعات first(x1),,first(xn)i تحتوي على حيث i<n فإن وفي حالة ماإذا كانت جميع المجموعات first(x1),,first(xn)i تحتوي على فإن(xه) first تحتوي أيضاعلى .

والآن وبعد الانتهاء من تعريف first(x)i نقوم بتعريف (first(i لأي سلسلة من الرموز النهائية واللانهائية عندما تكون x1x2xn = فإن () first تحتوي على first(x1)-{ }i وأيضا إذا كانت كل المجموعات first(x1),,first(xi)i تحتوي على عندما تكون i<n فإن() first تحتوي على { first(xi+1) i } وأهيرا إذا كانت first(xi)i تحتوي على لكل n,,i=1,2,3 فإن() first تحتوي أيضا على .

والتعريفات السابقة يمكن بسهولة تحويلها إلى خوارزمية لحساب المجموعات الأولى ولأن إيجاد المجموعة الأولى للرموز النهائية في غاية السهولة بينما تكمن الصعوبة فقط في حالة القيام بذلك للرموز اللانهائية لذلك فإن الخوارزمية التالية خاصة بحساب المجموعات الأولى للرموز اللانهائية فقط :
ويمكن أيضا أن نرى بسهولة ما تكون عليه الخوارزمية السابقة في حالة وجود أي إختيار بالقواعد النحوية يحتوي على ستصبح الخوارزمية أبسط كثيراً لأن First(A)i في هذه الحاله ستحتوي فقط على First(x1)i وبالتالي فإن الخوارزمية المبسطة ستصبح كما يلي:

; { }= (For all nonterminal A do First (A

While there are change to any first (A) do

For each production choice A x1, x2, x3... xn do

;(Add First(x1) to First (A

وعموما فإنه عند حساب المجموعة الأولى لأي رمز لا نهائي لا يتم فقط تحديد الرموز النهائية التي يمكن أن تظهر كرمز أول في سلاسل الحروف المشتقة من ذلك الرمز اللانهائي ولكن يتم أيضا تحديد ما إذا كانت السلسلة الفارغة E يمكن أن تشتق من هذا الرمز اللانهائي و الذي يطلق عليه في هذه الحالة رمز قابل للفراغ (Nullable) فأي رمز لا نهائي A يكون قابل للفراغ عند وجود اشتقاق على الشكل أي عندما يكون First(A)i تحتوي على .

و الآن نقدم مجموعة من الأمثلة لحساب المجموعات الأولى للرموز اللانهائية:

مثال 7

بفرض القواعد النحوية التالية الخاصة بالتعبير الرياضي البسيط والذي سبق أن عرضناه من قبل:

exp exp addop term | term

- | + Addop

term term mulop factor | factor

* Mulop

Factor (exp) | number

من الأسهل أن نعيد كتابة القواعد النحوية السابقة بحيث يكون كل اختيار في سطر منفصل مع ترقيم كل اختيار من تلك الإختيارات كما يلي:

exp exp addop term (1)

exp term (2)

+ addop (3)

- addop (4)

term term mulop factor (5)

term factor (6)

* mulop (7)

(factor (exp (8)

factor number (9)

وتلك القواعد النحوية لا تحتوي على أي اختيار يتضمن وبالتالي سنستخدم الخوارزمية المبسطة لحساب المجموعات الأولى ولكن قبل ذلك يجب ملاحظة أن كل من الاختيار رقم (1) والاختيار رقم (5) يحتويان على تكرار يساري وبالتالي فإن تلك القواعد النحوية لا تصلح للاستخدام في وضعها الحالي مع أسلوب الإعراب LL(1)i ولا يمكن أن نبني لها جدول الإعراب المستخدم في هذا الأسلوب ولكن هذا لا يمنع إمكانية حساب المجموعة الأولى للرموز اللانهائية التي تتضمنها تلك القواعد النحوية مع عدم الاستفادة من هذه الاختيارات التي تحتوي على تكرار يساري في هذه المهمة وذلك ما سيتضح عند حساب المجموعات الأولى والجدول التالي يوضح خطوات ذلك حيث يسجل الجدول فقط التغيرات التي تحدث في الخانة المقابلة للاختيار الذي أحدث ذلك التغير والخانات الفارغة تعني عدم حدوث تغير في هذه الخطوة.

ولا بد من إتمام دورة رابعة وأخيرة والتي لا تظهر في الجدول السابق لأنه لن يحدث بها أي تغير وبالتالي فإنه بعد أربع دورات ننتهي من حساب المجموعات الأولى التالية:

{ First (exp) = { (,number

{ First (term) = { (,number

{ First (factor) = { (,number

{ - , + } = (First (addop

{ * } = (First (mulop

مثال 8

بفرض القواعد النحوية التالية والخاصة بأمر if ولكن بعد إجراء التجميع اليساري عليها:

statement if - stmt │other

if - stmt if (exp) statement else - part

│ else - part else statement

exp 0│1

وهذه القواعد تتضمن أعلى اختيارات بها وذلك مع الرمز اللانهائي else - part فقط أي انه الرمز الوحيد المقابل للفراغ وبالتالي سيكون التعقيد فقط مع هذا الرمز وباقي الرموز اللانهائية لن تكون كذلك ولن تتأثر وهذا ما يحدث عادة في أغلب القواعد النحوية الخاصة بلغات البرمجة الفعلية حيث تكون

الاختيارات التي تتضمن محدودة للغاية وكالمثال السابق نبدأ بكتابة الاختيارات التي تحتوي عليها القواعد النحوية بشكل منفصل كالآتي:

statement if - stmt

Statement other

If - stmt if (exp) statement else - part

else - part else statement else -part

else - part e

exp 0

exp 1

والجدول التالي يوضح خطوات ودورات حساب المجموعة الأولى كما في المثال السابق:

وبعد إتمام الدورة الثالثة التي لا يحدث بها أي تغير نصل إلى المجموعات الأولى لتالية:

{First (statement)={ if, other

{ First (if-stmt)={ if

{First (else-port)={ else

{First (exp) = {0, 1

مثال 9

بفرض القواعد النحوية التالية الخاصة بمتابعة الأوامر:

'stmt-sequence stmt stmt-seq

| stmt-seq' ; stmt-seq

stmt S

ومرة أخرى نقوم بإعادة كتابتها ليكون كل اختيار في سطر منفصل كالأتي:

'stmt-sequence stmt stmt-seq

stmt-seq' ; stmt-seq

'Stmt-seq

Stmt S

وبعد ثلاثة دورات نحصل على المجموعات الأولى التالية:

{First (stmt-sequence) ={s

{ ,;} = (First (stmt-seq

{First (statement) = {S

المجموعات المتتالية

إذا كان A هو أحد الرموز اللانهائية فإن المجموعة Follow(A)i تتكون من رموز نهائية وأيضا الرمز $ في بعض الحالات وفقاً للقواعد التالية:

1) إذا كان A هو رمز البداية فإن $ تكون في Follow(A)i .

2) إذا كان هناك قاعدة نحوية مثل تكون في Follow(A)i

3) إذا كان هناك قاعدة نحوية مثل تحتوي على فإن Follow(A)i تحتوي على Follow(B)i .

ومن التعريف السابق للمجموعة المتتالية يمكن ملاحظة عدة نقاط:

* الأولى أن الرمز $ الذي يستخدم للإشارة إلى نهاية المدخلات يظهر في المجموعة التالية لرمز البداية وهذا أمر طبيعي لكون رمز البداية هو الذي يشتق منه جملة المدخلات وبالتالي فإن ما يتبع جملة المدخلات هو مؤشر نهاية المدخلات.

* النقطة الثانية أن السلسلة الفارغة لايمكن أن تظهر في المجموعة االتالية وذلك لكونها تظهر في المجموعة الأولى فقط ولا يمكن ظهورها ضمن مفردات جملة المدخلات.

* النقطة الثالثة فهي أن المجموعة التالية يتم حسابها فقط للرموز اللانهائية وذلك بخلاف المجموعة الأولى التي كان يتم حسابها للرموز النهائية وسلسلة الرموز النهائية واللانهائية أيضا وكان من الممكن ان يتم ذلك أيضا بالنسبة للمجموعى التالية ولكن ذلك غير ضروري وذلك لن إنشاء جدول الإعراب LL(1)i يحتاج المجموعات التالية الخاصة بالرموز اللانهائية فقط.

* النقطة الأخيرة ان تعريف المجموعة التالية يعتمد على الجانب الأيمن من القاعدة النحوية بعكس تعريف المجموعة الأولى الذي كان يعتمد على الجانب الأيسر من القاعدة النحوية وكقاعدة عامة فغن كل اختيار في قاعدة نحوية سيساهم في المجموعات التالية لكل رمز لأنها في الجانب الأيمن من القاعدة النحوية وهذا لا يحدث في حالة المجموعات الأولى لن كل اختيار في قاعدة نحوية سيضيف للمجموعة الأولى رمزاً لا نهائياً وحيداً فقط وهو الرمز اللانهائي الذي يظهر في الجانب الأيسر من القاعدة النحوية.

والآن نقدم الخوارزمية الخاصة بالمجموعات التالية تلك الخوارزمية التي تقوم بحساب المجموعات التالية وفقا للتعريف السابق لتلك المجموعات وسيكون كما يلي:

;{$}=: (Follow (start-symbol

;{ } = (For all nonterminal A start symbol do Follow (A

While there are changes to any follow sets do

For each production A X1 X2 Xn do

For each Xi that is a nonterminal do

;(Add First (Xi+1 Xi+2 Xn) - { } to Follow (Xi

If is in First (Xi+1 Xi+2 Xn) then

;(Add Follow (A) to Follow (Xi

وبالطبع وكما حدث بالنسبة للمجموعات الأولى فإنه يمكن تبسيط الخوارزمية السابقة في حالة عدم وجود قواعد نحوية تحتوي على السلسلة الفارغة ونقدم الآن ثلاثة أمثلة لحساب المجموعات التالية وهي نفس الأمثلة التي سبق أن تم حساب المجموعة الأولى لها.

مثال 10

مرة أخرى بفرض القواعد النحوية الخاصة بالتعبير الرياضي البسيط والتي تم حساب المجموعات الأولى لها في مثال رقم 7 والتي كانت كما يلي:

{first(exp) = {(, number

{first(term) = {(, number

{first(factor) = {(, number

{-,+}=(first(addop

{*}=(first(mulop

ونعيد الآن كتابة اختبارات تلك القواعد النحوية بشكل منفصل كالأتي:

exp exp addop term (1)

exp term (2)

+ addop (3)

- addop (4)

term term mulop factor (5)

term factor (6)

* mulop (7)

(factor (exp (8)

Factor number (9)

حيث نلاحظ أن الاختبارات النحوية رقم (3)،(4 (، (7)، (9) لا تحتوي على رموز لانهائية في الجانب الأيمن للقاعدة النحوية وبالتالي فإنها لن تضيف شيئاً عند حساب المجموعات المتتالية وسنستخدم فقط باقي الإختيارات بالترتيب على أن نبدأ بإضافة الرمز $ للمجموعة المتتالية الخاصة بالرموز اللانهائية الأخرى فإنها ستبدأ فارغة وبعد ذلك عند استخدام الاختيار رقم 1 فإنه سيؤثر على المجموعات المتتالية الخاصة بثلاثة رموز لا نهائية هيُ exp، addop، term حيث سيتم إضافة First،(addop) إلى المجموعة المتتالية ل exp وبالتالي ستصبح كما يلي:

{- ,+ ,$} = (Follow (exp

وكذلك سيتم إضافة First(term)i إلى المجموعة المتتالية لـ addop مما يجعلها تصبح كالآتي:

{Follow (addop) = {(, number

وأخيرا سيضاف المجموعة المتتالية exp إلى المجموعة المتتالية ل exp على المجموعة المتتالية لـ term لتصبح كما يلي:

{-,+ ,$} = (Follow (term

وعند استخدام الخيار رقم 2 سيتم إضافة المجموعة المتتالية ل exp إلى المجموعة المتتالية term وهو ما سبق أن تم مع الاختيار رقم 1 السابق وبالتالي فلن تتغير حالة المجموعة المتتالية الخاصة بالرمز اللانهائي term .

بينما عند استخدام الاختيار رقم 5 سيتم إضافة First(mulop)i إلى المجموعة المتتالية ل term لتصبح كما يلي:

{*,- ,+ ,$} = (Follow (term

ثم سيتم إضافة First(factor)i إلى المجموعة المتتالية ل mulop لتصبح كالأتي: Follow(mulop)={(, number} i

وأخيرا سيتم إضافة المجموعة المتتالية ل term إلى المجموعة المتتالية ل Factor لتصبح كالأتي:

{*,-,+ ,$} = (Follow (factor

ولن يؤثر استخدام الاختيار رقم 6 على حالة المجموعة المتتالية الخاصة بالرمز اللانهائي factor بينما سيؤدي استخدام الاختيار رقم (8) إلى إضافة first(')')i على المجموعة المتتالية لـ exp لتصبح كالتالي :

{(, - ,+ , $ } = (Follow (exp

وعند ذلك تنتهي الدورة الأولى لحساب المجموعات المتتالية للرموز اللانهائية وبعدها تبدأ الدورات التي تليها وهكذا حيث يعرض الجدول الآتي ملخصاً لدورات حساب تلك المجموعات المتتالية موضحاً التغييرات التي ستحدث عند استخدام كل اختيار من اختيارات القواعد النحوية بعد استبعاد الاختبارات رقم 3 , 4, 7 , 9 التي لن تؤثر على حساب المجموعات المتتالية:

ولن يحدث أي تغيير في الدورة الثالثة التي لم تظهر في الجدول السابق وبالتالي سينتهي حساب المجموعات التالية للرموز اللانهائية وستكون كما يلي في وضعها الأخير:

{(,_,+, $ } = (Follow (exp

{Follow(addop)={ (, number

{(,*,-,+,$} =(Follow(term

{Follow(mulop)={(,number

{(,*,-,+,$ }=(Follow(factor

مثال 11

أيضا بفرض القواعد النحوية الخالية الخاصة بأمر if مرة أخرى والذي سبق وتم حساب المجموعات الأولى بها في المثال رقم 8 وكانت كما يلي:

{First(statement)={if, other

{First(if stmt)={if

{, First(else-part)={else

{First(exp)={0,1

وبإعادة كتابة اختيارات القواعد النحوية بشكل منفصل كالآتي:

statement →if stmt (1)

statement→other (2)

if stmt→if (exp) statement else part (3)

else part →else statement (4)

→ else part (5)

exp → 0 (6)

exp→1 (7)

ونلاحظ أن الاختيارات رقم (2) , (5) ,(6) ,(7) لا تحتوي في الجانب الأيمن على أي رموز لا نهائية لذلك فإنها لن تضيف شيء عند حساب المجموعات المتتالية وسنستخدم فقط باقي الاختيارات بالترتيب على أن نبدأ بإضافة الرمز $ إلى المجموعة المتتالية الخاصة برمز البداية اللانهائي statement بينما باقي المجموعات المتتالية الخاصة بالرموز اللانهائية الأخرى فإنها ستبدأ فارغة وبعد ذلك عند استخدام الاختيار رقم (1) سيتم إضافة المجموعات المتتالية لـ statement إلى المجموعات المتتالية للرمز اللانهائي if stmt لتصبح كما يلي:

{$}= (Follow(if stmt

وعند استخدام الاختيار رقم (3) سوف يؤثر ذلك على المجموعات المتتالية الخاصة بالرموز اللانهائية else-part, statement, exp حيث يتم إضافة first(')')i إلى المجموعة المتتالية لـ exp لتصبح follow(exp)={)}i ثم يضاف first(else-part)-{}i إلى المجموعة المتتالية لـ statement لتصبح كالآتي:

{Follow(statement) ={$,else

وأخيرا يتم إضافة المجموعة المتتالية لـ if- stmt إلى المجموعة المتتالية ل

else part

وأيضا إلى المجموعة المتتالية ل statement ليصبحا كما يلي:

{$}=(Follow (else- part

{Follow(statement)={$,else

ويلاحظ عدم حدوث تغيير على المجموعة المتتالية لـ statement وعند استخدام الاختيار رقم (4) فإن المجموعة المتتالية لـ else part يتم أضافتها إلى المجموعة المتتالية statement والذي لن يحدث أي تغيير عليها وبذلك تنتهي الدورة الأولى لحساب المجموعات المتتالية للرموز اللانهائية لتبدأ الدورة

الثانية باستخدام الإختياررقم (1) حيث تتم إضافة المجموعة المتتالية لـ statement إلى المجموعة المتتالية ل (if-stmt) لتصبح كما يلي:

{Follow(if-stmt)={$,else

أما استخدام الاختيار رقم (3) فإنه سيؤدي إلى إضافة المجموعة المتتالية

لـ if stmt لتصبح كالآتي:

{Follow(else- part)= {$,else

وأخيرا لن يغير استخدام الاختيار رقم (4) أي شيء بالنسبة للمجموعة المتتالية لـ statement وبذلك تنتهي الدورة الثانية لتبدأ الدورة الثالثة التي لن تحدث أي تغير على المجموعات المتتالية للرموز اللانهائية حيث تكون تلك المجموعات المتتالية كما في وضعها الأخير :

{Follow(statement)={$,else

{Follow(if-stmt)={$,else

{Follow(else - part)={$,else

{(}=(Follow(exp

مثال12:

في هذا المثال نقوم بحساب المجموعات المتتالية للقواعد النحوية الآتية الخاصة بتتابع الأوامر والتي سبق تقديمها في المثال رقم (9) :

'stmt - sequence → stmt stmt - seq (1)

stmt - seq' → ; stmt - sequence (2)

→ 'stmt - seq (3)

Stmt → S (4)

وفي المثال رقم (9) تم حساب المجموعات الأولى و كانت كما يلي:

{ First (stmt - sequence) ={ s

{Frits (stmt) = { S

{First (stmt - seq') ={; , E

والإختيارات رقم (3) و (4) لن تؤثر على حساب المجموعات المتتالية حيث تستخدم باقي الإختيارات ونبدأ بإضافة الرمز $ الى المجموعة المتتالية ل stmt - sequence وباقي المجموعات المتتالية ستبدأ فارغة وبعد انتهاء الدورة الأولى ستكون المجموعات المتتالية كما يلي:

{$} = (Follow (stmt _ sequence

{;} = (Follow (stmt

{$}=('Follow { stmt _ seq

ولن يحدث أي تغير في الدورة الثانية وبالتالي ستبقى المجموعات المتتالية كما هي.
بناء جدول الإعراب LL(1)i

نشرح الآن كيفية بناء جدول الإعراب المستخدم في طريقة الإعراب LL(1)i والسابق الإشارة اليه في الأجزاء السابقة من هذه الوحدة حيث يتم البناء وفقاً للقاعدتين الأتيتين:

1. إذا كان هو إحدى إختيارات القواعد النحوية وهناك الإشتقاق حيث a هي أحد المفردات ففي هذه الحاله تتم اضافة إلى خانة جدول الإعراب M[A , a]i

2. إذا كان هو إحدى إختيارات القواعد النحوية وهناك الإشتقاق

وكذلك الإشتقاق حيث S هو رمز البداية و a هي احدى المفردات ففي هذه الحاله تتم إضافة إلى خانة جدول الإعراب M[A , a]i

وواضح أن المفردة a تكون في القاعدة الأولى في ال () First بينما في القاعدة الثانية تكون في ال Follow(A)i وبذلك نصل إلى الخوارزم الأتي الخاص ببناء جدول الإعراب :LL(1)i

و الآن بعض الأمثلة الخاصة ببناء جدول الإعراب للقواعد النحوية السابق التعرض لها في الأمثلة السابقة من هذه الوحدة.

مثال 13:

بفرض القواعد النحوية الخاصة بالتعبير الرياضي البسيط الذي تم استخدامه أكثرمن مرة خلال هذه الوحدة وحيث أن هذه القواعد في شكلها الأصلي تتضمن على تكرار يساري فقد تم حذف التكرار اليساري وأصبحت كما يلي :

'exp term exp

exp' addop term exp' | Є

- | + addop

'term factor term

term' mulop factor term' | Є

* mulop

factor (exp) | number

ولا بد أن نقوم بحساب المجموعات الأولى والتالية للرموز اللانهائية التي تحتوي عليها تلك القواعد النحوية وسوف نترك القيام بهذه الحسابات كتمرين للقارئ على أن نقدم النتيجة النهائية فقط لهذه المجموعة والتي ستكون كما يلي:

{ First (exp) = { (,number

{ First (exp') ={ + , - , Є

{- , + } = (First (addop

{ First (term) = { (,number

{ First (term') = { * , Є

{ * } = (First (mulop

{ First (factor) = { (,number

{ (, $ } = (Follow (exp

{ (, $} = ('Follow (exp

{ Follow (addop) = { (, number

{ -, +, (, $ } = (Follow (term

{ -, +, (, $ } = ('Follow (term

{ Follow (mulop) = { (, number

{ -, +, (, $ } = (Follow (factor

والآن نستطيع بناء جدول الإعراب LL(1)i لتلك القواعد النحوية والذي سيكون كما يلي:

حيث يلاحظ أن هذا الجدول هو نفس الجدول الذي تم عرضه من قبل عند شرح طريقة الإعراب LL(1)i .

مثال 14:

بفرض القواعد النحوية المبسطة لأمر if الآتية:

statement if - stmt | other

if - stmt if (exp) statement else - part

else - part else statement | Є

exp 0 | 1

والتي كانت المجموعات الأولى والتالية الخاصة بها كما يلي :

{ First (statement) = { if , other

{ First (if - stmt) = { if

{ First (else - part) = { else , Є

{ First (exp) = { 0 , 1

{ Follow(statement)= { $, else

{ Follow(if _ stmt)= { $, else

{ Follow(else - part)= { $, else

{ (} =(Follow(exp

و بالتالي يمكننا بناء جدول الإعراب LL(1)i لتلك القواعد النحوية والذي سيكون كما يلي:

وهذا الجدول أيضا هو نفس الجدول السابق عرضه من قبل شرح طريقة الإعراب LL(1)i .

مثال 15 :

بفرض القواعد النحوية الأتية الخاصة بتتابع الأوامر والتي سبق تقديمها من قبل:

'stmt - sequence stmt stmt - seq

stmt - seq' stmt - sequence | Є

stmt s

والتي كانت المجموعات الأولى والتالية الخاصة بها كالآتي :

{First(stmt - sequence)= {s

{ First(stmt - seq')= {; Є

{First(stmt)= {s

{$} =(Follow(stmt - sequence

{$} = ('Follow(stmt - seq

{$, ; } =(Follow(stmt

وجدول الإعراب LL(1)i الخاص بتلك القواعد النحوية سيكون كما يلي :

. الإعراب التنازلي التكراري للغة الافتراضية

نناقش في هذا الجزء الإعراب التنازلي التكراري للغة الإفتراضية التي سبق تقديمها في الوحدة الأولى كما تم في الوحدة السابقة عرض للقواعد النحوية الخاصة بها ووحدة الاعراب التي نقدمها تقوم بإنشاء شجرة النحو بالإضافة إلى طباعة هيكل هذه الشجرة في ملف يمكن استعراضه بعد ذلك.

والإعراب التنازلي التكراري يستخدم الشكل EBNF الآتي وهو المقابل للقواعد النحوية الخاصة بتلك اللغة الإفتراضية التي سبق عرضها باستخدام الشكل BNF في الوحدة السابقة:

program stmt - sequence

{stmt - sequence statement {; statement

{statement if - stmt │repeat - statement

assign - stmt │read - stmt │write - stmt│

If - stmt if exp then stmt - sequence

else stmt - sequence] end]

repeat - stmt repeat stmt - sequence until exp

assign -stmt read identifier

write - stmt write exp

[exp simple - exp[comparison - op simple - exp

= │ > comparison - op

{simple - exp term {addop term

- │ + addop

{term factor {mulop factor

/ │ * mulop

factor (exp) │ number │ identifier

ووحدة إعراب اللغة الافتراضية تتبع الإطار العام للإعراب التنازلي التكراري السابق شرحه في بداية هذه الوحدة وستتكون من احدى عشر إجراء تكراري تقابل مباشرة القواعد النحوية السابقة حيث يوجد إجراء للرمز اللانهائي Stmt-Sequence و آخر للرمز Statement وخمسة إجراءات تقابل مستويات الأولويات الخاصة بالتعبير الرياضي. وليس من الضروري إعداد إجراءات مستقل للرموز اللانهائية التي تمثل العمليات ويكتفي بالتعرف عليها كجزء من التعبير المرتبطة به ونفس الشيء بالنسبة للرمز اللانهائي Program حيث أنه مجرد تتابع من الأوامر والذي يمثلها الرمز Stmt - Sequence الذي يمثله إجراء بالفعل وبالتالي فإن وحدة الإعراب الخاصة باللغة الإفتراضية تبدأ باستدعاء ذلك الإجراء مباشرة.

ووحدة الإعراب تتضمن أيضا المتغير Lookahead الذي يحتفظ بالمفردة الحالية أثناء تحليل الصيغ النحوية وكذلك الإجراء match لتنفيذ التطابق مع مفردة محددة على أن يقوم باستدعاء GetToken في حالة حدوث هذا التطابق للحصول على المفردة التاليه والا فإنه يظهر رسالة خطا باستخدام اجراء خاص لذلك وليكن اسمه Syntax Error الذي يقوم بطباعة رسالة الخطأ.

وتبدأ عملية الإعراب بتخزين أول مفردة بملف المدخلات داخل المتغير Lookahead ثم يتم استدعاء الإجراء Stmt - Sequence الذي يقوم باختبار نهاية ملف المدخلات قبل إرجاع شجرة النحو الذي تم بناءها والتي يمكن اعداد اجراء خاص لطباعة نسخة خطية منها.

. الاستعفاء من الأخطاء

إن رد فعل وحدة الاعراب تجاه الأخطاء النحوية (Syntax Error) والتي تقاس بمدى قدرة وحدة الإعراب على الإستعفاء من الاخطاء (Error Recovery) تعتبر غالبا من العوامل الأساسية التي تحدد مدى امكانية استخدام المترجم والحد الأدنى عند تحليل الصيغ النحوية هو تحديد هل البرنامج سليم من الناحية النحوية أم لا وذلك من طريق تميز سلاسل الحروف للغة خالية السياق التي يتم اشتقاقها من القواعد النحوية الخاصة بلغة البرمجة المكتوب بها البرنامج وفي حالة وجود بعض الأخطاء النحوية في البرنامج فإنه لا بد أن يشير لمثل هذه الأخطاء.

وبجانب هذا الحد الأدنى المطلوب من عملية تحليل الصيغ النحوية فإن هناك عدة مستويات لرد الفعل تجاه هذه الأخطاء وعموما تحاول وحدة الإعراب أن تعطي رسالة خطأ واضحة على الأقل لأول خطأ يتم اكتشافه مع محاولة تحديد مكان حدوث الخطأ بقدر الإمكان و بعض وحدات الإعراب تذهب بعيدا بمحاولة إيجاد طريقة لتصحيح الخطأ الذي تم اكتشافه بشكل أو بآخر ولكن في الواقع هذا لا يمكن إلا في حالة الأخطاء البسيطة والشائعة وخلاف ذلك فإن أغلبية المحاولات لا تكون ذات كفاءة عالية وتختلف كثيرا عما كان ينوي المبرمج القيام به ولهذا فإن مهمة تصحيح الأخطاء نادراً ما تقوم بها وحدة الإعراب ولكن في الواقع تكون هناك صعوبة لتوليد رسالة خطأ واضحة دون الخوض في محاولة تصحيح هذا الخطأ.

وعموما فإن أغلب أساليب الاستعفاء من الأخطاء تكون فردية ولا يمكن تعميمها بمعنى أنها مرتبطة بلغة برمجة محددة وخاصة بخوارزمية إعراب محددة مع العديد من الحالات الخاصة لمواقف منفصلة مع صعوبة وجود مبادئ عامة ولكن مع ذلك هناك بعض النقاط الهامة التي يتم أخذها في الإعتبار عادة والتي تتمثل في الآتي:

1- إن وحدة الإعراب يجب أن تحاول إكتشاف الخطأ مبكرا بقدر الإمكان لأن التأخير في ذلك يؤدي إلى صعوبة تحديد نوع ومكان حدوث هذا الخطأ.

2- بعد حدوث الخطأ يجب على وحدة الإعراب أن تحدد المكان المحتمل لإستكمال عملية الإعراب حتى يمكن إعراب أكبر جزء من البرنامج بقدر الإمكان وذلك من أجل إكتشاف العديد من الأخطاء الحقيقية خلال عملية ترجمة واحدة.

3- يجب على وحدة الإعراب أن تتفادى مشكلة تتابع الأخطاء (Error Cascade) والتي تحدث عندما يتتبع حدوث خطأ معين سلسلة من الرسائل المرتبطة بنفس الخطأ.

4- يجب على وحدة الإعراب تفادي الدوائر اللانهائية من الأخطاء (Infinite Loops) والتي تحدث تتابع غير منتهي من رسائل الأخطاء دون وجود مدخلات جديدة.

وبعض الأهداف السابقة تتعارض مع بعضها البعض لذلك فإن القائم بإعداد أي مترجم لا بد أن يأخذ هذا في الحسبان عند بناء الجزء الخاص بالتعامل مع الأخطاء.
تحـــــليل الدلالات

Semantic Analysis
. الخصائص والقواعد الخصائصية

إن خصائص تراكيب لغات البرمجة تختلف اختلافاً كبيراً من لغة إلى أخرى في المعلومات التي تحتوي عليها ومدى تعقيدها وأيضاً في الوقت الذي يتم تحديدها فيه وهل هو أثناء مراحل ترجمة البرنامج أم تشغيله وعموماً فإن ما يلي أمثلة معتادة لمثل هذه الخصائص:

- نوع بيانات المتغير.

- قيمة التعبير.

- موقع المتغير في ذاكرة الحاسب.

- شفرة الإجراء بعد الترجمة.

- عدد الأرقام في العدد.

وتلك الخصائص قد تكون ثابتة أثناء عملية الترجمة أو تمثل الحد الأدنى أو الأقصى فمثلاً أرقام العدد قد يكون عددها ثابت أو يكون له حد أدنى أو أقصى حسب تعريف لغة البرمجة. وأيضاً يمكن أن يكون تحديد قيمة الخصائص فقط أثناء التنفيذ مثل تحديد قيمة التعبير الذي يحتوي على متغيرات أو تحديد موقع تراكيب البيانات ذات التخصيص المتحرك (المتغير) (Dynamic Allocation) وعموماً فان عملية حساب أي خاصية وإلحاق قيمتها المحسوبة بتركيبة اللغة هو ما يطلق عليه الـ Binding الخاص بالخاصية. وتوقيت حدوث ذلك يسمى الـ (Binding Time) والذي يكون مختلفاً بالنسبة للخصائص المختلفة ولنفس الخاصية قد يكون مختلفاً من لغة برمجة إلى أخرى. وإذا كان يتم أثناء عملية الترجمة وقبل البدء في التنفيذ تسمى الخاصية خاصية ساكنة (Static Attributes) أما إذا كان خلاف

ذلك فيطلق عليها خاصية متحركة (Dynamic Attribute) وبالطبع فإن معدي المترجمات يهتمون فقط بالخصائص الساكنة.

والآن وعند النظر لأمثلة الخصائص السابقة لمعرفة توقيت تحديد كل منها ومدى أهمية ذلك أثناء عملية الترجمة نجد ما يلي:

- في لغات البرمجة التي يتم فيها تحديد النوع بشكل ساكن مثل لغة السي والـ Pascal فإن نوع بيانات المتغير يعتبر أحد الخصائص الساكنة الهامة وعملية اختبار النوع (Type checking) تتم أثناء مرحلة تحليل الدلالات حيث يتم تحديد قيمة خاصية نوع البيانات لكل مكونات اللغة التي يتم تعريف نوع بياناتها والتحقق منها وفقاً لقواعد اللغة. ولكن في لغة مثل الـ Lisp والتي يتم فيها تحديد أنواع البيانات بشكل متحرك فإن المترجم يكون عليه توليد شفرة لتحديد الأنواع واختبارها أثناء تنفيذ البرنامج.

- أن قيم التعبيرات عادة ما تكون متحركة حيث يقوم المترجم بتوليد شفرة لحساب تلك القيم أثناء تنفيذ البرنامج ولكن بعض التعبيرات التي تحتوي على قيم ثابتة فقط (مثل 3 + 4 * 5) فان محلل الدلالات يختار حسابها لإيجاد قيمة التعبير النهائية أثناء الترجمة.

- أن تخصيص موقع المتغيرات في الذاكرة أما أن يكون ساكن أو متحرك وذلك حسب اللغة وصفات المتغير ذاته فمثلاً في لغة الـ FORTRAN فإن تخصيص جميع المتغيرات يكون ساكن بينما في لغة الـ Lisp يكون تخصيص جميع المتغيرات متحركاً. أما في لغة الـ C والـ Pascal فإن تخصيص المتغيرات يكون خليطاً بعضها ساكن والبعض الآخر متحرك حسب نوع المتغير وعادة ما يؤجل المترجم الحسابات المرتبطة بتخصيص المتغيرات حتى مرحلة توليد شفرة الهدف حيث أن تلك الحسابات ترتبط ببيئة تشغيل البرنامج وتفاصيل الحاسب المستهدف.

- إن شفرة الهدف لأي إجراء هي بالتأكيد أحد الخصائص الساكنة وتوليد شفرة الهدف هي مرحلة المترجم المعنية بهذه الخاصية.

- إن عدد أرقام العدد هي أحد الخصائص التي غالباً ما يتم معالجتها بشكل ضمني أثناء عملية الترجمة وذلك لكونها مرتبطة بطريقة تمثيل القيم والذي عادة ما يكون جزءًا من بيئة تشغيل البرنامج ولكن محلل المفردات قد يحتاج إلى معرفة عدد الأرقام المسموح بها لكي يكون العدد صحيح.

وكما شاهدنا من الأمثلة السابقة فإن حسابات الخصائص تختلف بشكل كبير وعندما تظهر في المترجم فإن ذلك قد يحدث في أي وقت أثناء عملية الترجمة ولكن مع ذلك يتم ربط حسابات الخصائص بمرحلة تحليل الدلالات على الرغم من كون بعض الخصائص يكون هناك احتياج لمعلومات عنها أثناء مرحلة تحليل المفردات أو مرحلة تحليل الصيغ النحوية وسوف نركز فقط في هذه الوحدة على حسابات الخصائص التي تتم بعد الانتهاء من الإعراب وقبل البدء في توليد شفرة الهدف من المترجم

1 القواعد الخصائصية

إن الخصائص ترتبط مباشرة برموز القواعد النحوية الخاصة باللغة سواء الرموز النهائية أو اللانهائية فإذا كان X هو أحد هذه الرموز و a هي إحدى الخصائص المرتبطة بـ X فإننا نكتب X.a للتعبير عن قيمة الخاصية a المرتبطة بالرمز X . وطريقة الكتابة هذه تشابه الطريقة المستخدمة في

التعبير عن حقول السجلات في لغة الـ Pascal وعناصر التراكيب (Structure) في لغة السي وذلك على الرغم من أن الطريقة المعتادة لتطبيق حسابات الخصائص هو وضع قيمة الخصائص داخل عقد شجرة النحو باستخدام حقول السجلات أو عناصر التراكيب كما سيتضح في هذا الجزء.

وعادة ما يتم كتابة القواعد الخصائصية بحيث يكون مع كل قاعدة نحوية القاعدة أو القواعد الدلالية المرتبطة بها كما في الشكل التالي:

القواعد الدلالية Semantic Rules	القاعدة النحوية	Grammar Rule
القواعد الدلالية المرتبطة بالقاعدة الأولى	القاعدة الأولى	Rule 1
القواعد الدلالية المرتبطة بالقاعدة الثانية	القاعدة الثانية	Rule 2
.	.	.
.	.	.
القواعد الدلالية المرتبطة بالقاعدة الأخيرة	القاعدة الأخيرة	Rule n

ولأننا سبق أن شرحنا القواعد الخصائصية بالتفصيل في الوحدة الثانية من هذه المادة العلمية لذلك سنقدم مباشرة بعض الأمثلة الإضافية لتلك القواعد:

مثال 1

بفرض القواعد النحوية البسيطة التالية والخاصة بالأعداد الصحيحة بدون الإشارة:

number number digit | digit

digit 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

فإن أهم خاصية للعدد هي قيمته والتي سنعطيها الاسم val وكل رقم له قيمة تتوافق مباشرة مع الرقم الفعلي الذي يمثله فمثلاً القاعدة النحوية digit 3 تعنى أن الرقم له القيمة 3 في هذه الحالة وهو ما يمكن التعبير عنه بالقاعدة الدلالية:

digit.val = 3

لذلك سنربط تلك القاعدة الدلالية بالقاعدة النحوية digit 3 ونفس الشيء بالنسبة لباقي الأرقام وكذلك بالنسبة للأعداد فكل عدد له قيمة حسب الأرقام التي يحتوي عليها فإذا كان العدد يشتق باستخدام القاعدة النحوية

number digit

فإن العدد يتكون في هذه الحالة من رقم وحيد وبالتالي فإن قيمته هي قيمة هذا الرقم والقاعدة الدلالية التي تعرض هذه الحقيقة تكون كما يلي :

number.val = digit.val

أما إذا كان العدد يحتوي على أكثر من رقم فإنه يشتق باستخدام القاعدة النحوية:

number number digit

ومن الضروري توضيح العلاقة بين قيمة الرمز الموجود في الجانب الأيسر من القاعدة النحوية وقيم الرموز التي تظهر في الجانب الأيمن من القاعدة النحوية مع ملاحظة أن الرمز number يظهر مرتان في القاعدة النحوية لذلك يجب التفرقة بينهما لأن قيمة الرمز number الذي في الجانب الايمن سوى الأيسر وسنفرق بين الاثنين باستخدام الترقيم وسنعيد كتابة القاعدة النحوية لتصبح كما يلي:

number1 number2 digit

وبافتراض عدد مثل 34 فإن الاشتقاق من أقصى اليسار لهذا العدد يكون كالآتي:

number number digit digit digit 3 digit 34

وبفرض استخدام القاعدة النحوية الأخيرة في أول خطوات الاشتقاق فإن الرمز اللانهائي number2 سيقابل الرقم 3 بينما الرمز اللانهائي digit سيقابل الرقم 4 وستكون القيمة التي تقابل كلاً منهما هي 3 و 4 على التوالي. وللوصول إلى قيمة الرمز number1 يجب ضرب قيمة الرمز number2 في 10 ثم إضافة قيمة الرمز digit وذلك كما يلي :

number1.val = 3 * 10 + 4 = 34

بمعنى ترحيل القيمة 3 خانة عشرية واحدة جهة اليسار مع إضافة القيمة 4 وهذا يقابل القاعدة الدلالة التالية:

number1. val = number2. val * 10 + digit

وبالتالي نصل إلى القواعد الخصائصية التالية للخاصية val :

ومعنى القواعد الدلالية لسلسلة محددة من الأرقام يمكن توضيحه أكثر باستخدام شجرة الإعراب فمثلاً شجرة الإعراب الخاصة بالعدد 345 تكون كما يلي :

ومعنى القواعد الدلالية لسلسلة محددة من الأرقام يمكن توضيحه أكثر باستخدام شجرة الإعراب فمثلاً شجرة الإعراب الخاصة بالعدد 345 تكون كما يلي:

وفي الشكل السابق نلاحظ أن الحسابات التي تقابل القواعد الدلالية المحددة تظهر تحت العقد الداخلية من شجرة الإعراب وعرض القواعد الدلالية كحسابات في شجرة الإعراب يكون هاماً بالنسبة لخوارزميات حساب قيم الخصائص وذلك ما سنتعرض له في الجزء القادم.

مثال 2

بفرض القواعد النحوية التالية الخاصة بالتعبيرات الرياضية البسيطة:

exp exp + term │ exp − term │ term

term term * factor │ factor

factor (exp) │ number

والخاصية الأساسية للرموز اللانهائية exp و term و factor هي قيمتها الرقمية والتي يمكن أن تكتب val لتصبح القواعد الدلالية المرتبطة بالقواعد النحوية كما يلي:

والقواعد الدلالية السابقة تظهر العلاقة بين الصيغ النحوية للتعبيرات الرياضية وبين الدلالات الخاصة للحسابات الرياضية الواجب إنجازها ويلاحظ عدم وجود قاعدة دلالية يظهر فيها number.val في الجانب الأيسر لها مما يعني ضرورة حساب قيمة هذه الخاصية قبل أي تحليل دلالي يمكن أن يستخدمها. وليكن ذلك في مرحلة تحليل المفردات والحل الآخر البديل لذلك هو إضافة قواعد نحوية وقواعد دلالية (مثل الموضحة في المثال السابق) تمكن من حساب قيمة تلك الخاصية.

والآن نستطيع أن نظهر الحسابات الخاصة بتلك الخاصية عن طريق إضافة معادلات القواعد الدلالية لعقد شجرة الإعراب وبغرض التعبير الرياضي (34 - 3) * 42 فإننا يمكن توضيح دلالات قيمة هذا التعبير باستخدام شجرة الإعراب التالية:

مثال 3

بفرض القواعد النحوية التالية والخاصة بتعريف المتغيرات (Variables Declaration) في لغة السي مع بعض التبسيط:

decl type var-list

type int | float

var-list id, var-list | id

ونريد تحديد نوع البيانات (Data Type) كخاصية مصاحبة للمتغيرات التي تم تقديمها من خلال المعرفات (Identifiers) في القواعد النحوية السابقة مع كتابة القواعد الدلالية التي تحدد كيف تكون تلك الخاصية مرتبطة بنوع التعريف الخاص بالمتغيرات. وسنقوم بذلك من خلال إعداد القواعد الخصائصية المتعلقة بالخاصية **dtype** (التي بالطبع تختلف عن الرمز اللانهائي **type** في القواعد النحوية) حيث ستكون تلك القواعد الخصائصية كما يلي:

ونلاحظ في القواعد الدلالية السابقة أن قيمة الخاصية **dtype** هي إما **integer** أو **real** والتي تقابل المفردات **int** و **float** على التوالي وأن الرمز اللانهائي **type** له أيضاً قيمة للخاصية **dtype** يتم تحديدها بواسطة المفردة التي يمثلها وهذه القيمة هي التي تحدد قيمة الخاصية لجميع المتغيرات التي يعبر عنها بالرمز اللانهائي **var-list** وهو ما يتضح من القاعدة الدلالية المصاحبة للقاعدة النحوية الخاصة بالرمز اللانهائي **decl** كما أن قيمة الخاصية **dtype** لكل **id** في قائمة المتغيرات **var-list** تحدد وفقاً لقيمتها لدى قائمة المتغيرات **var-list** نفسها كما يلاحظ أن الرمز اللانهائي **decl** ليس لديه الخاصية **dtype** لعدم الاحتياج لذلك وهو ما يعني عدم ضرورة إلحاق أي خاصية بجميع الرموز النهائية واللانهائية التي تحتوي عليها القواعد النحوية.

وكالسابق نقدم الآن شجرة الإعراب التي تعرض القواعد الدلالية بفرض أن سلسلة المدخلات كما يلي:

float   x, y

فإن شجرة الإعراب الخاصة بها ستكون كالتالي:

ويلاحظ أن جميع الأمثلة السابقة كانت تحتوي على خاصية واحدة فقط ولكن ذلك ليس ضرورياً في جميع الحالات. فقد تتضمن القواعد الخصائصية عدة خصائص مترابطة في نفس الوقت. والمثال التالي يقدم حالة بسيطة تحتوي على عدة خصائص يربط بينها علاقة.

مثال 4

بفرض إجراء تعديل على القواعد النحوية الخاصة بالأعداد الصحيحة والتي سبق استخدامها في المثال رقم (1) وهذا التعديل يسمح بأن يكون العدد إما عدداً عشرياً (Decimal) أو عدداً ثمانياً (Octal) على أن يتم تحديد ذلك باستخدام حرف واحد يلي العدد فإذا كان الحرف o فهذا يعني أن العدد ثماني أما إذا كان الحرف d فهذا يعني أن العدد عشري وعند ذلك فإن القواعد النحوية ستكون كالآتي:

based-num num basedchar

basedchar o | d

num num digit | digit

digit 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

. الخوارزميات الخاصة بحساب الخصائص

في هذا الجزء سندرس طرق استخدام القواعد الخصائصية كأساس للمترجم عند حساب واستخدام الخصائص المعّرفة بواسطة القواعد الدلالية والتي تقوم على تحويل تلك القواعد الدلالية إلى إجراءات حسابية وعند النظر إلى المعادلة التي تمثل كل قاعدة دلالية على كونها أمر تخصيص لقيم الخصائص الموجودة في الجانب الأيمن للأمر إلى الخاصية الموجودة في الجانب الأيسر فإن نجاح هذا الأمر يتطلب أن تكون جميع الخصائص التي تظهر في الجانب الأيمن قد سبق حسابها من قبل ولكن هذا المطلب يتم إهماله في القواعد الخصائصية وذلك لأن معادلات القواعد يمكن أن يتم سردها بأي ترتيب دون أن يؤثر ذلك على صحتها. ولهذا فإن المشكلة الأساسية عند إعداد الخوارزمية المقابلة للقواعد الخصائصية تكمن في إيجاد ترتيب سليم لتطبيق معادلات القواعد الدلالية الذي يضمن نجاح أوامر التخصيص المقابلة لتلك المعادلات بحيث تكون جميع قيم الخصائص المستخدمة في الحسابات متوفرة عند إنجاز تلك الحسابات وعموما فإن المعادلات تحدد شروط الترتيب المناسب لإجراء الحسابات الخاصة بالخصائص التي تحتوي عليها لذلك فإن المهمة الأولى هي إظهار تلك الشروط المتعلقة بالترتيب وذلك باستخدام أحد المخططات الموجهة (Directed Graph) والتي تسمى مخطط الإعتمادية (Dependency Graph).

1.2. المخططات الإعتمادية والترتيب

إن كل اختيار في القواعد النحوية التي تشملها القواعد الخصائصية يكون له مخطط اعتمادي مرتبط به وهذا المخطط يحتوي على عدة عقد وكل عقدة منهم تكون معنونه بكل خاصية Xi.aj لكل رمز نهائي أو لانهائي في تلك القاعدة النحوية بالإضافة إلى أن لكل معادلة بالقواعد الدلالية على الشكل :

(,xm.ak,)Xi.aj = fij

مرتبطة بهذه القاعدة النحوية يكون هناك رابط (edge) من كل عقدة Xm.ak في الجانب الأيمن للمعادلة الدلالية إلى العقدة Xi.aj وهذا الرابط هو الذي يوضح أن الخاصية Xi.aj تعتمد على خاصية Xm.ak وبالتالي فلا بد أن يتم حساب قيمة الخاصية Xm.ak حتى يمكن حساب قيمة الخاصية Xi.aj

وبالتبعية فإن لأي سلسلة حروف في اللغة يمكن توليدها باستخدام قواعد نحوية خالية السياق يكون لها مخطط اعتمادي يتكون من اتحاد المخططات الإعتمادية لاختيارات القواعد النحوية التي تمثل جميع عقد شجرة الإعراب الخاصة بسلسلة الحروف.

وعند رسم المخطط الإعتمادي لكل قاعدة نحوية أو لسلسلة الحروف فإن العقد المرتبطة بكل رمز X يتم رسمها في مجموعات بحيث تظهر الروابط كهيكل حول شجرة الإعراب وهذا ما سنوضحه من خلال الأمثلة التالية.

مثال 5

بفرض القواعد النحوية التي في المثال رقم (1) مع القواعد الخصائصية التي تم إعدادها في ذلك المثال والتي كانت تتضمن على خاصية وحيدة وهي val لهذا فإن كل رمز نهائي أو لا نهائي يكون له مخطط اعتمادي يتكون من عقدة واحدة تمثل تلك الخاصية مع هذا الرمز فمثلا القاعدة النحوية:

number1 → number2 digit

لها قاعدة دلالية واحدة مرتبطة بها وهي:

number1.val = number2.val * 10 + digit.val

لذلك فإن المخطط الاعتمادي لهذه القاعدة النحوية يكون كالآتي:

و بنفس الطريقة فإن المخطط الاعتمادي للقاعدة النحوية number → digit يكون كمايلي:

وبالنسبة لباقي القواعد النحوية فإن المخططات الاعتمادية الخاصة بها ستكون في غاية البساطة لكونها لا تحتوي على أي روابط لكون حساب قيمة الخاصية digit.val سيتم مباشرة من الجانب الأيمن لكل قاعدة نحوية.

وفي النهاية فإن المخطط الاعتمادي الخاص بالعدد 345 سيكون الآتي:

وهذا المخطط الأخير هو ما يقابل شجرة الإعراب الخاصة بهذا العدد والتي تم تقديمها في المثال رقم (1)

مثال 6

بفرض القواعد النحوية في المثال رقم 3 والقواعد الخصائصية التي تم إعدادها والمتعلقة بالخاصية dtype وكانت القاعدة النحوية:

var- list1 → id , var‐list2

يرتبط بها قاعدتين دلاليتين هما:

id-dtype = var-list1.dtype

var-list2.dtype = var-list1.dtype

لذلك فإن المخطط الاعتمادي لها سيكون كما يلي:

وبنفس الطريقة فإن المخطط الاعتمادي للقاعدة النحوية var-list → id سيكون كالتالي:

بينما المخطط الاعتمادي لكل من القاعديتن type→int , type → float فسيكونان في غاية البساطة لعدم وجود روابط بهما وفي النهاية فإن المخطط الاعتمادي للقاعدة النحوية:

decl → type var‐list

والتي يرتبط بها القاعدة الدلالية:

var-list.dtype = type.dtype

سيكون كما يلي:

وفي هذه الحالة الأخيرة ولكون الرمز اللانهائي decl لا يظهر في المخطط الإعتمادي فإن ارتباط ذلك المخطط بالقاعدة النحوية يكون غير واضح ولهذا السبب ولعدة أسباب أخرى فإن المخطط الاعتمادي غالبا ما يتم رسمه حول جزء شجرة الإعراب المتعلق بالقاعدة النحوية وبالتالي فإن المخطط الاعتمادي السابق يمكن إعادة رسمه كالتالي:

وهو ما يجعل القاعدة النحوية التي يرتبط بها المخطط الإعتمادي واضحة ولكن يلاحظ عدم كتابة الخاصية مع الرمز اللانهائي بالشكل المعتاد في هذا الوضع الأخير للمخطط الإعتمادي بل يتم كتابة الخاصية بجوار العقدة التي تمثل الرمز اللانهائي وبالتالي فإن المخطط الاعتمادي الأول في هذا المثال عند إعادة رسمه مع شجرة الإعراب سيكون كما يلي :

وفي النهاية فإن المخطط الاعتمادي لسلسلة الحروف float x, y سيكون كالشكل التالي:

مثال 7

بفرض القاعدة النحوية التي في المثال رقم 4 مع القواعد الخصائصية المتعلقة بالخاصيتين base, val وسوف نقوم برسم المخططات الإعتمادية لأربعة قواعد نحوية فقط هي:

based-num → num basechar

num → num digit

num → digit

digit → 9

ونبدأ بالمخطط الإعتمادي للقاعدة النحوية الأولى والتي سيكون كما يلي:

وهذا المخطط السابق يمثل القاعدتين الدلاليتين المرتبطتين بالقاعدة النحوية الأولى أما القاعدة النحوية الثانية فإن المخطط الاعتمادي الخاص بها سيكون كالتالي:

والمخطط الإعتمادي للقاعدة النحوية الثالثة فإنها سيكون كما يلي:

وأخيرا سيكون المخطط الإعتمادي للقاعدة النحوية الأخيرة كالأتي :

وعند رسم المخطط الاعتمادى للعدد 3450 فانه سيكون كمايلى:

وبفرض أننا نريد الآن تحديد الخوارزمية التي تقوم بحساب خصائص القواعد الخصائصية باستخدام معادلات القواعد الدلالية كقواعد حسابية في سلسلة المفردات المحددة والمراد ترجمتها سيقوم المخطط الإعتمادي لشجرة الإعراب الخاص بها بتحديد قيود الترتيب التي يجب أن تتبعها الخوارزمية لكي تتمكن من حساب خصائص تلك السلسلة من المفردات وبالطبع فإن أي خوارزمية يجب أن تقوم بحساب قيمة الخاصية في كل عقدة من عقد المخطط الإعتمادي قبل أن تحاول حساب قيمة الخاصية التي تليها فى ترتيب الاعتمادية فلابد ان يتم حساب الخاصية ai قبل الخاصية aj إذا كانت الخاصية aj تعتمد على الخاصية ai وترتيب التنقل في المخطط الإعتمادي هو الذي يحدد قيود الترتيب التي يجب أن تلتزم بها الخوارزمية وعموما ولكي ينجح ذلك فإنه لابد ألا يكون هناك أي دورات في المخطط الإعتمادي أي يشترط في المخطط أن يكون مخططا موجهاً لا دائرياً (Dissected Acyclic Graph) والذي يعرف بالاسم المختصر DAG .

مثال 8

إن المخطط الاعتمادي الذي في المثال السابق والخاص بالرقم 3450 هو مخطط من النوع DAG وسيتم ترقيم العقد التي يحتوي عليها وفقا لترتيب التنقل الذي يمكن أن يتبعه الخوارزمية في حساب قيمة الخصائص حيث سيكون المخطط الإعتمادي كما يلي بعد الترقيم مع حذف شجرة الإعراب من أجل سهولة الرؤية :

ويجب التأكيد على أن ترتيب التنقل الذي يوضحه ترقيم العقد في الشكل السابق ليس هو الترتيب الوحيد للتنقل الذي يجب أن تلتزم به خوارزمية حساب الخصائص فيمكن أيضا التنقل داخل المخطط الإعتمادي السابق بالترتيب التالي من اليمين إلى اليسار :

14 13 10 7 5 4 8 3 11 2 1 9 6 12

وكما شاهدنا فإنه من الممكن أن تعتمد الخوارزمية الخاصة بتحليل الخصائص على بناء المخطط الإعتمادي أثناء الترجمة بحيث يحدد ترتيب التنقل داخل المخطط الترتيب الذي يتم به حساب الخصائص ولأن بناء المخطط الإعتمادي يقوم على شجرة إعراب محددة في وقت الترجمة لذلك يطلق على هذه الطريقة في بعض الأحيان طريقة شجرة الإعراب (Parse Tree Method) وهذه الطريقة ليست هي الطريقة الوحيدة لحساب قيم الخصائص التي تتضمنها القواعد الخصائصية فهناك طريقة أخرى بديلة تعرف بالطريقة القائمة على القواعد (Rule- Based Method) وهذه الطريقة الأخيرة تعتمد على قيام معد المترجم بتحليل القواعد الخصائصية لتحديد ترتيب مناسب لحساب قيم الخصائص التي تحتوي عليها القواعد الخصائصية. هذا الترتيب هو الذي سيستخدم عند بناء المترجم وعموماً فإن تحديد الترتيب المناسب يتوقف على نوعية الخصائص المطلوب حساب قيمتها وهذا هو موضوع الجزء القادم.

. الخصائص المصنعة والموروثة

إن حساب قيمة الخصائص القائم على تحليل القواعد النحوية يعتمد أيضا على ترتيب التنقل داخل شجرة الإعراب أو شجرة النحو وهذا الترتيب يختلف حسب نوع ارتباط الخصائص ببعضها البعض وحتى نستطيع معرفة الفرق بين أنواع الارتباط سنقوم بتصنيف الخصائص وفقاً لنوع الارتباط ونبدأ بالنوع الأول البسيط والذي تتميز به الخصائص المصنعة (Synthesized Attributes) حيث تعتبر الخاصية مصنعة إذا كانت جميع ارتباطتها في شجرة الإعراب من أسفل إلى أعلى أي من الابن إلى الأب فإذا كانت القاعدة النحوية على الشكل التالي:

$$A \rightarrow X1\ X2\ X3\ .\ .\ .\ Xn$$

فلابد أن تكون جميع معادلات القواعد الدلالية المرتبطة والتي تظهر فيها الخاصية a في الجانب الأيسر من المعادلة على الشكل الآتي:

$$(A.a = F(X1.a1, . . . , X1.ak , . . . , Xn.a1 . . . , Xn.ak$$

لكي تكون تلك الخاصية a مصنعة ونحن بالفعل شاهدنا عدداً من الأمثلة التي تحتوي على خصائص مصنعة فالخاصية val في المثال رقم (1) هي خاصية مصنعة وكذلك الخاصية val في المثال رقم (2) فهي أيضا خاصية مصنعة وفي حالة هذا النوع من الخصائص وعند الانتهاء من بناء شجرة الإعراب أو شجرة النحو في مرحلة تحليل الصيغ النحوية فإنه يمكن حساب قيم ذلك النوع من الخصائص أثناء التنقل مرة واحدة من أسفل إلى أعلى داخل الشجرة أو التنقل ذو الترتيب اللاحق (Postorder Traversal) لتلك الشجرة وهذا ما يوضحه شفرة الإجراء التكراري التالي :

```
(procedure PostEval (T : treenode
begin
for each child C of T do
;(PostEval(C
;compute all synthesized attributes of T
;end
```

وبالطبع ليست جميع الخصائص مصنعة فهناك أيضا الخصائص الموروثه (Inherited Attributes) وهي تلك الخصائص التي لا تتوفر فيها الشروط السابقة للخصائص المصنعة وقد شاهدنا أيضا في الأمثلة السابقة بعض الخصائص الموروثة مثل الخاصية dtype في المثال رقم (3) وكذلك الخاصية base في المثال رقم (4) وهذا النوع من الخصائص يكون فيه الارتباط من أعلى إلى أسفل في شجرة الإعراب أي من الأب إلى الابن أو يكون ألارتباط بين الأبناء المشتركين في نفس الأب ويمكن توضيح هذين النوعين من الارتباط بالشكلين التاليين:

وهذان النوعان من الإرتباط موجودان في الخاصية dtype في المثال رقم (6) وعموما فإن الخصائص الموروثة يمكن حساب قيمتها بالتنقل أيضا داخل شجرة الإعراب أو شجرة النحو ولكن في هذه الحالة بالتنقل ذو الترتيب السابق (Preorder Traversal) وشفرة الإجراء التكراري التالي توضح ذلك :

```
(procedure PreEval (T : treenode
begin
for each child C of T do
;compute all inherited attributes of C
;(PreEval(C
;end
```

وعلى عكس الخصائص المصنعة فإن ترتيب حساب الخصائص الموروثة عند الأبناء مهم وذلك لإمكانية وجود ارتباط بين خصائص تلك الأبناء كما أوضحنا في السطور السابقة لذلك لا بد أن يكون حساب الخصائص الموروثة لدى الأبناء متناسب مع حالة الارتباط بينهم.

3. جدول الرموز

يعتبر جدول الرموز (Symbol Table) أهم الخصائص الموروثة في المترجم كما أنه أهم تراكيب البيانات المستخدمة بعد شجرة النحو وعلى الرغم من تأجيل شرح جدول الرموز إلى هذه المرحلة الخاصة بتحليل الدلالات الا انه يجب التاكيد أن جدول الرموز متضمن أيضا في المراحل السابقة لهذه المرحلة حيث يظهر دوره الفعال في مرحلة تحليل الصيغ النحوية وكذلك مرحلة تحليل المفردات سواء عند الاحتياج لإضافة بيانات إلية او عند استشارته لحل الغموض الذى يحدث فى بعض الحالات. ومع هذا وفي حالة لغات البرمجة المصممة بدقة شديدة مثل لغة ال Pascal ولغة ال Ada فإنه قد يتم تأجيل التعامل مع جدول الرموز لما بعد الانتهاء من تحليل الصيغ النحوية وعند التأكد من أن البرنامج المطلوب ترجمته سليم نحويا وهذا بالفعل ما تم بالنسبة للغة الافتراضية التي سيتم دراسة جدول الرموز الخاص يها في نهاية الوحدة.

وعموما فإن لجدول الرموز ثلاثة عمليات أساسية هي الإضافة (Insert) والبحث (Lookup) والحذف (Delete) هذا بالإضافة إلى بعض العمليات المساعدة الأخرى وعملية الإضافة تستخدم لتخزين معلومات تم الحصول عليها من خلال تعريفات الأسماء (Name Declaration) . أما عملية البحث فنحتاج إليها لاسترجاع المعلومات الخاصة باسم محدد عند ورود هذا الإسم في شفرة البرنامج بينما عملية الحذف فيكون الاحتياج لها عندما يصبح تعريف أحد الأسماء ليس له ضرورة. وبصفة عامة فإن صفات تلك العمليات الثلاثة يحكمها قواعد لغة البرمجة المطلوب ترجمتها وبالنسبة للمعلومات التي يتم تخزينها في جدول الرموز فهي حسب هيكل الجدول والغرض من التعريفات. ولكن عادة ما تشمل معلومات عن نوع البيانات (Data Type) والمدى (Scope) الذي يكون خلاله الاسم صالحاً للاستخدام وكذلك الموقع في الذاكرة (Memory Location) وسوف ندرس في هذا الجزء تنظيم هيكل بيانات جدول الرموز وفي النهاية نقدم مثالا شاملا لاستخدام جدول الرموز مع أحد القواعد الخصائصية.

1.3 هيكل جدول الرموز

إن هيكل جدول الرموز مثل القاموس وكفاءة عملياته الأساسية (الإضافة والبحث والحذف) تختلف حسب تنظيم هيكل البيانات الخاص به وعموما فإن تحليل كفاءة التنظيمات المختلفة وفحص الأساليب التنظيمية الجيدة تعتبر من الموضوعات الأساسية لمادة هياكل البيانات لهذا فلن نقوم بدراستها بالتفصيل في هذه المادة العلمية وسنكتفي بإعطاء نظرة عامة فقط لأهم هياكل البيانات التي تصلح للإستخدام مع جدول الرموز عند بناء المترجم والتي تشمل القوائم الخطية (Linear Lists) وهياكل شجرة البحث (Search Trees) المختلفة مثل شجر البحث الثنائى (Binary Search Trees) وكذلك الجداول العشوائية (Symbol Table).

والقوائم الخطية هى إحدى هياكل البيانات البسيطة والجيدة فى نفس الوقت التي يمكنها تنفيذ العمليات الثلاثة الأساسية لجدول الرموز بسهولة وبأسلوب مباشر حيث تتم عملية الإضافة سواء في بداية أو نهاية القائمة في وقت ثابت بينهما عمليتي البحث والحذف فإن الوقت الخاص بها يتوقف على حجم القائمة ولهذا فإن القوائم الخطية قد تكون جيدة بشكل كافي لبناء جدول الرموز الخاص بالمترجمات التي لا يمثل فيها سرعة الترجمة عنصراً مهماً بالنسبة لها مثل المترجمات التجريبية والمترجمات الفورية

للبرامج صغيرة الحجم. أما هياكل شجر البحث فهي أقل استخداما بالنسبة لجدول الرموز لكونها ليست الأعلى كفاءة بالإضافة إلى تعقيد عملية الحذف. بينما الجداول العشوائية فهي تعتبر أفضل الاختيارات لتنفيذ جدول الرموز ذلك لكون جميع عمليات الجدول الأساسية يمكن تنفيذها في وقت ثابت إلى حدٍ كبير ولهذا فهي تستخدم بكثرة في بناء جدول الرموز.

2.3. التعريفات

إن أداء جدول الرموز يرتبط بقوة بالصفات الخاصة بالتعريفات (Declarations) في اللغة المطلوبة ترجمتها فمثلا كيفية تفعيل عمليتي الإضافة إلى والحذف من جدول الرموز وتوقيت الاحتياج لهذه العمليات والخصائص التي تضاف إلى الجدول تختلف كثيرا من لغة إلى أخرى هذا بالإضافة إلى أن مرحلة بناء جدول الرموز خلال عملية الترجمة والفترة الزمنية التي نحتاج فيها إلى بقائة قد يختلف أيضا من لغة إلى أخرى وفي هذا الجزء سوف نشير إلى بعض النقاط المتعلقة بالتعريفات فى لغة البرمجة التى تؤثر على اداء وتنفيذ جدول الرموز وبصفة عامة هناك أربعة أنواع أساسية من التعريفات التي تحدث بكثرة في لغات البرمجة وهي تعريفات الثوابت (Constant Declarations) وتعريفات الأنواع (Type Declarations) وتعريفات المتغيرات (Variable Declarations) وأيضا تعريفات الإجراءات والوظائف (Procedure / Function Declarations) .

وتعريفات الثوابت تشمل ما يلي في لغة Pascal :

;Const int SIZE = 200

وتعريفات الأنواع تتضمن الآتي في لغة ال Pascal :

; type table = array[1SIZE] of Entry

وأيضا تعريف struct في لغة السي مثل :

struct entry

} ; char * name

; int count

;struct entry * next

;{

كما أن لغة السي لديها الأسلوب typedef تعريف أسماء بديلة للأنواع مثل:

;typedef struct entry * entryptr

أما بالنسبة لتعريفات المتغيرات وهي أكثر أنواع التعريفات شيوعا في أغلب لغات البرمجة فهي تشمل الآتي في لغة أل FORTRAN :

(integer a, b(100

وكذلك ما يلي في لغة السي:

;[int a, b[100

وأخيرا هناك تعريفات الإجراءات والوظائف مثل تعريف الوظيفة الآتية في لغة السي :

(int factorial (int num

}

;int fact = 1

(--for(int i=num; i>=1 ; i

;fact *= i

;return fact

{

حيث أنها لا تختلف عن تعريف لثابت من نوع إجراء أو وظيفة ولكن أغلب اللغات ولطبيعته الخاصة تعتبره نوعاً مستقلاً من التعريفات.

وجميع أنواع التعريفات تكون واضحة (Explicit) لوجود أمر خاص باللغة تستخدم في تلك التعريفات ويمكن أيضا أن تكون التعريفات ضمنية (Implicit) عن طريق إلحاقها بالأوامر التنفيذية كما هو الحال في لغة ال basic و FORTRAN اللتان تسمحان باستخدام المتغيرات بدون تعريف ظاهر وفي هذ الحالة الأخيرة فإن الأمر يتطلب بعض الجهد من أجل تجميع نفس المعلومات التي يتم الحصول عليها في حالة التعريف الظاهر. ففي لغة ال FORTRAN مثلا لديها قاعدة هي أن جميع المتغيرات التي تبدأ بالحروف من I وحتى N وليس لها تعريفاً ظاهراً تعتبر أنها متغيرات صحيحة (Integer) أما إذا بدأت بأي حروف أخرى فإنها تعتبر متغيرات حقيقية (Real) وفي بعض الأحيان يطلق على التعريفات الضمنية

أنها تعريفات بالاستخدام (Declaration by Use) من كون أول استخدام للمتغير الذي ليس له تعريف ظاهر كما لو كان يتضمن تعريف هذا المتغير بشكل ضمني.

وغالباً ومن أجل السهولة يتم استخدام جدول رموز واحد للاحتفاظ بالأسماء من مختلف أنواع التعريفات وخاصة أن جميع لغات البرمجة لا تسمح باستخدام نفس الإسم في أنواع التعريفات المختلفة ولكن في بعض الأحوال يفضل استخدام أكثر من جدول للرموز مثل تخصيص جدول منفصل للرموز لكل نوع من أنواع التعريفات على حدة. وبعض اللغات تفضل إلحاق جدول رموز منفصل مع المناطق المختلفة التي يتضمنها البرنامج على أن يتم ربطها معاً وفقاً للقواعد الدلالية الخاصة باللغة.

هذا ويختلف ربط الخصائص بالاسم من خلال التعريف وفقاً لنوع التعريفات. فتعريفات الثوابت تلحق القيم بالأسماء وتلك القيم هي التي تحدد كيف يعالجها المترجم وبالطبع فإن تخصيص القيمة للثوابت يتم مرة واحدة. وبمجرد تحديد قيمتها فإنه لا يمكن تغييرها بعد ذلك. وتعريف الثوابت قد تتضمن أيضاً تحديد نوع البيانات الخاص بكل ثابت بشكل ظاهر مثلها في ذلك مثل المتغيرات بينما لا يتم ذلك في لغة ال Pascal الذي يتم فيها تحديد نوع البيانات حسب القيمة التي يتم تخصيصها للثابت.

وفي تعريفات الأنواع يتم تحديد اسم النوع الجديد الذي يتم تعريفه وقد يتم إنشاء اسم بديل للنوع السابق تعريفه وعادة ما تستخدم أسماء الأنواع مع الخوارزمية المتعلقة بتوافق الأنواع (Type Equivalence) لإنجاز الاختبارات الخاصة بالأنواع وفقا لقواعد اللغة.

وبالنسبة لتعريفات المتغيرات فإنها غالبا ما يتم فيها ربط الأسماء بأنواع البيانات وقد يتم أيضا تحديد بعض الخصائص بشكل ضمني مثل المدى (Scope) الذي يحدد منطقة البرنامج ولكنه قد يتأثر بالتفاعلات مع التعريفات الأخرى ويجب التأكد أن المدى قد يكون أيضا أحد خصائص الثوابت والأنواع والإجراءات وليس فقط المتغيرات. لهذا سيتم مناقشته بالتفصيل في جزء مستقل.

وأحد الخصائص الأخرى المرتبطة بمدى المتغير الذي يتم تعريفه هي موقع الذاكرة المخصص لذلك المتغير وفترة صلاحية هذا التخصص ففي لغة السي على سبيل المثال فإن جميع المتغيرات التي يتم تعريفها خارج الوظائف يكون التخصيص لموقع الذاكرة الخاص بهذه المتغيرات ثابت (Static) وبالتالي فإن فترة صلاحيته يكون خلال فترة تشغيل هذه الوظيفة فقط مع العلم بأن لغة السي تسمح بتوسيع مدى التعريف الذي يتم داخل أي وظيفة وجعله ثابتاً وذلك بإضافة كلمة Static إلى التعريف كما في الوظيفة التالية:

```
(int count (void
}
;static int count = 0
;return ++count
{
```

وعموما فإننا لن نهتم بدراسة أساليب تخصيص مواقع الذاكرة في هذه المادة العلمية وبدلا من هذا سنعود لتحليل المدى والأساليب المتعلقة بمعالجة المدى في جدول الرموز وذلك في الجزء التالي.

. قواعد المدى

تختلف القواعد المتعلقة بالمدى في لغات البرمجة المختلفة اختلافا كبيرا ولكن هناك العديد من القواعد الشائعة في العديد من اللغات وسنركز في هذا الجزء على قاعدتين فقط من القواعد الشائعة للمدى الأولى مرتبطة بالتعريف قبل الاستخدام بينما تتعلق القاعدة الثانية بالتفرع الأقرب في الأوامر المركبة. قاعدة التعريف قبل الاستخدام (Declaration Before Use) هي قاعدة شائعة ومستخدمة في لغة السي و ال Pascal و التي تتطلب أن يتم تعريف الاسم داخل البرنامج قبل أي استخدام لهذا الإسم وهذه القاعدة تتطلب بناء جدول الرموز أثناء مرحلة تحليل الصيغ النحوية بحيث تتم عملية البحث عن الاسم داخل جدول الرموز مع كل استخدام له وإذا فشلت عملية البحث في إيجاد الإسم فهذا يعني حدوث مخالفة للقاعدة التي نحن بصددها. وعند ذلك يقوم المترجم بإظهار رسالة خطأ مناسبة وهذه القاعدة تدعم عملية الترجمة من مرحلة واحدة وهناك بعض لغات البرمجة لا تشترط القيام بالتعريف قبل الاستخدام. وتلك اللغات تحتاج لمرحلة منفصلة لبناء جدول الرموز وبالتالي لايمكن إجراء الترجمة من مرحلة واحدة. أما فيما يتعلق بالهياكل الكتلية (Block Structure) الشائعة الاستخدام في لغات البرمجة الحديثة والتي تقوم على استخدام تراكيب برمجية تحتوي على تعريفات تسمى كتلة (Block) وهو ما ينطبق في لغة البرمجة Pascal على البرنامج الرئيسي (Main Program) . وعلى التعريفات الخاصة بالإجراءات والوظائف وفي لغة السي تشمل وحدات الترجمة (Compilation Units) وتعريفات الوظائف والأوامر المركبة التي يتم وضعها بين الأقواس { } ، بينما في لغات البرمجة الموجه للأغراض (Object-Oriented Programming Language) فتنطبق على تعريفات الفصائل (Class Declarations) . وعموماً فإنه يطلق على لغة البرمجة أنها لغة مهيكلة كتلية (Block Structured Language) إذا سمحت بتفرع كتل داخل كتل أخرى على أن يكون مدى التعريفات داخل أي كتلة محصوراً في هذه الكتلة فقط وجميع الكتل الأخرى التي يتضمنها وعند وجود أكثر من تعريف مختلف لنفس الإسم فإن التعريف الذي يطبق عند استخدام الإسم هو التعريف الذي يقع في أقرب كتلة فرعية لهذا الاستخدام وهذه القاعدة هي ما تسمى قاعدة التفرع الأقرب (Most Closely Nested Rule) ولتوضيح الهياكل المجمعة ومدى تأثير قاعدة التفرع الأقرب على جدول الرموز نقدم جزء برنامج السي التالي:

```
;int i , j

(int f (int size

;char i , temp }

...

; double j }



{
```

; char * j }

{

{

حيث يلاحظ أنه يحتوي على خمسة كتل الكتلة الأولى يشمل جميع الخطوات ويحتوي على تعريفات المتغيرات الصحيحة i , j و الوظيفة f و الكتلة الثانية هي تعريف الوظيفة f نفسها والتي تحتوي على تعريف المتغير size أما الكتلة الثالثة فهو الأمر المركب الخاص بجسم الوظيفة f والتي يحتوي على تعريفات المتغيرات الرقمية الحرفية i , temp (ويمكن اعتبار الكتلتين الثانية والثالثة ككتلة واحدة) بينما الكتلة الرابعة فهي الأمر المركب الذي يحتوي على تعريف المتغير j كمتغير حقيقي مزدوج وأخيرا الكتلة الخامسة وهو أمر مركب أيضا يحتوي على تعريف المتغير j كمؤشر حرفي وداخل الوظيفة f يوجد تعريفاً واحداً لكل من المتغير size والمتغير temp في جدول الرموز وبالتالي فإن أي استخدام لهذين المتغيرين سيكون وفقا لهذا التعريف الوحيد لهما وبالنسبة للمتغير i فإنه يوجد تعريف محلي (Local Declaration) له كمتغير حرفي داخل جسم الوظيفة هذا بالإضافة إلى تعريف آخر له غير محلي كمتغير صحيح ولكن وفقا لقاعدة التفرع الأقرب فإن التعريف المحلي هو الذي يؤخذ به في هذه الحالة وبنفس الطريقة بالنسبة للمتغير j الذي يوجد له ثلاثة تعريفات أحدها غير محلي قبل الوظيفة f وتعريفان محليان داخل كل من الكتلة الرابعة والخامسة. ووفقا لنفس القاعدة فإن التعريف المحلي الخاص بكل مجمع منهما هو الذي سيطبق داخلهما ولن يعاد تطبيق التعريف غير المحلي إلا بعد الخروج من الوظيفة f

وفي بعض اللغات مثل أل Pascal و أل Ada يتم السماح بالتفرع في الإجراءات والوظائف أي يمكن أن يكون هناك إجراء أو وظيفة داخل إجراء أو وظيفة أخرى وهذا يزيد من التعقيد في بيئة تشغيل البرنامج المكتوب بأحدى هاتين اللغتين. وعموما فإنه يجب عند تنفيذ عملية الإضافة إلى جدول الرموز عدم فقد التعريفات السابقة للأسماء ولكن يتم فقط إخفاءها بحيث لا يظهر أثناء عملية البحث عن اسم محدد إلا التعريف الأحدث فقط لهذا الاسم. وكذلك بالنسبة لعملية الحذف حيث لا يجب حذف جميع التعريفات الخاصة باسم محدد ولكن أحدثها فقط ليتم الكشف عن أحد التعريفات السابقة لهذا الإسم بمعنى أن يتم تنفيذ عمليات الإضافة لجميع الأسماء المعرفة عند دخول كل كتلة على أن يتم تنفيذ عمليات الحذف المقابلة لنفس الأسماء عند الخروج من الكتلة وذلك ما يعني أن جدول الرموز يعمل مثل المكدس (Stack) أثناء التعامل مع تفرعات المدى.

وهناك أيضا عدد من الطرق الأخرى البديلة التي يمكن استخدامها للتعامل مع تفرعات المدى وإحدى هذه الطرق هو بناء جدول رموز جديد لكل مدى على أن يتم ربطها معا من المدى الداخلي إلى المدى الخارجي بالترتيب بحيث تستطيع عملية البحث أن تستمر أوتوماتيكياً في البحث داخل الجدول التالي عند الفشل في إيجاد الإسم في الجدول الحالي. وفي هذه الطريقة فإن الخروج من مدى معين يحتاج إلى جهد أقل لأنه بدلا من حذف جميع التعريفات التي تمت داخل هذا المدى باستخدام عملية الحذف فإنه يمكن حذف الجدول بالكامل بجميع محتوياته في خطوة واحدة.

4.3. تفاعل تعريفات نفس المستوى

أحد المواضيع المرتبطة بالمدى هو التفاعل (Interaction) بين التعريفات التي تتم في نفس المستوى من التفرع أي في كتلة واحدة وهذا التفاعل يختلف وفقاً لنوع التعريف وأيضا حسب اللغة المطلوب ترجمتها ولكن هناك مطلب معتاد في أغلب لغات البرمجة وهو عدم استخدام نفس الإسم في أكثر من تعريف داخل نفس مستوى التفرع فمثلا إذا تمت التعريفات التالية في لغة C فإنها تحدث خطأ في الترجمة :

```
typedef int i;
int i;
```

ولاختبار هذا المطلب فإن المترجم لا بد أن يقوم بعملية البحث داخل جدول الرموز قبل كل عملية إضافة على أن يحدد بأسلوب معين إذا كان هناك تعريفاً سابقاً بنفس الاسم في نفس مستوى التفرع أم لا ولكن السؤال الأكثر صعوبة في هذا الخصوص ما هو حجم المعلومات المتاحة لكل تعريف عن التعريفات الأخرى الموجودة معه في نفس المستوى من التفرع؟ فمثلا بفرض جزء البرنامج التالي المكتوب بلغة C :

```
int i = 1;
void f (void)
{ int i = 2, j = i + 1;
}
```

والسؤال الان ما هي القيمة المبدئية التي ستخزن في المتغير j الذي داخل الوظيفة f وهل هي 2 أم 3 وهذا يتوقف على أيهما سيستخدم التعريف المحلي للمتغير i أم التعريف غير المحلي لهذا المتغير وقد يكون من الطبيعي أن يتم استخدام التعريف المحلي على اعتبار أنه التعريف الأحدث وكذلك تطبيقا لقاعدة التفرع الأقرب التي شرحناها في الجزء السابق. وفي الحقيقة هذا ما يتم في لغة السي وهذا بافتراض ان كل تعريف يتم إضافته على حدة على جدول الرموز أثناء تنفيذ أمر التعريف المتعدد وهذه الطريقة في تنفيذ التعريفات تسمى طريقة التعريفات المتتابعة (Sequential Declarations) لأنه من الممكن بدلا من ذلك أن يتم إضافة جميع التعريفات بشكل متزامن ومرة واحدة إلى جدول الرموز في نهاية تنفيذ الجزء الخاص بالتعريفات وهذه الطريقة تسمى طريقة التعريفات المتوازية (Collateral Declarations) وفي هذه الحالة فإن استخدام أي اسم داخل تعبيرات خلال الجزء الخاص بالتعريفات

سيكون وفقا للتعريف السابق له وليس التعريف الجديد وهناك بعض لغات البرمجة تتبع تلك الطريقة في التعريفات والتي يتطلب إتباعها عدم إضافة التعريف مباشرة على جدول الرموز الحالى ولكن تجميعها فى جدول مؤقت ثم اضافتها بعد ذلك إلى جدول الرموز عند الانتهاء من جميع التعريفات.

وهناك أيضا حالة التعريفات التكرارية (Recursive Declarations) والتي يشير فيها التعريف السابق إلى نفسه والتي تحدث عند تعريف الإجراءات أو الوظائف التكرارية (Recursive Functions) حيث تستدعي الوظيفة نفسها كما في الوظيفة التالية المكتوبة بلغة السي والتي تقوم بحساب القاسم المشترك الأعظم (Greatest Common Divisor) لعددين صحيحين:

```
(int gcd (int n, int m
;if (m==0) return n }
;(else return gcd(m, n % m
{
```

ولكي تتم ترجمة الوظيفة السابقة بنجاح فإنه يجب على المترجم إضافة اسم الوظيفة gcd إلى جدول الرموز قبل دخول جسم الوظيفة وإلا فإنه لن يجد هذا الاسم في الجدول عند الوصول إلى الجزء التكراري الذي تستدعي فيه الوظيفة نفسها والحالة الأكثر تعقيدا هي حالة وجود مجموعة من الوظائف يتم بينها التكرار بشكل غير مباشر كما في الجزء التالي من البرنامج بلغة السي:

```
(void f (void
{ g }
(void g (void
{ f }
```

ففي تلك الحالة لن يكون كافياً إضافة اسم الوظيفة قبل الدخول في جسمها بل إن جزء البرنامج السابق سوف يحدث خطأ في الترجمة عند استدعاء الوظيفة g داخل الوظيفة f لان الوظيفة g لن يكون قد تم إضافتها بعد إلى جدول الرموز عند استدعائها داخل الوظيفة f و قد تم حل هذه المشكلة في لغة السي من خلال تعريف ما يسمى نموذج الوظيفة (Function Prototype) وفي الحالة السابقة يتم تعريف نموذج للوظيفة g قبل الوظيفة f كما يلي:

```
/* void g(void); /* function prototype
(void f(void
```

{ ()g }

(void g(void

{ ()f }

ونتيجة لتعريف نموذج للوظيفة g ، فقد تم توسيع المدى الخاص لهذا الإسم لتشمل الوظيفة f وذلك لأن المترجم سيقوم بإضافة اسم الوظيفة إلى جدول الرموز عند تعريف النموذج الخاص بها وقبل البدء في تعريف الوظيفة نفسها وبالطبع لا بد أن يحدث تطابق بين نموذج الوظيفة وتعريفها في عدد ونوع المعطيات التي تحتاجها.
. مثال شامل للقواعد الخصائصية باستخدام جدول الرموز

نريد الآن تقديم مثال يعرض عدداً من صفات التعريفات التي تم وصفها مع تطوير القواعد الخصائصية التي تجعل هذه الصفات واضحة في أداء جدول الرموز. والقواعد النحوية التي ستستخدم في هذا المثال هي القواعد النحوية الخاصة بالتعبيرات الرياضية البسيطة بعد إضافة الجزء الخاص بالتعريفات إليها لتصبح كالتالي:

S → exp

exp → (exp) │exp + exp │id │num

let dec�list in exp │

dec�list → dec�list , dec │dec

dec → id = exp

وهذه القواعد النحوية تتضمن الرمز اللانهائي S كرمز للبداية ولأن القواعد الخصائصية تحتوي على خصائص موروثة ستحتاج إلى تحديد قيمتها في البداية عند جذر شجرة النحو. لذلك سنكتفي بعملية حسابية واحدة فقط هي عملية الجمع من أجل التبسيط. ويلاحظ أن هناك غموضاً في تلك القواعد النحوية ولكن نفترض أن وحدة الإعراب قد قامت بالفعل بحل هذا الغموض وتم إنشاء شجرة النحو.

ومثل الأمثلة السابقة باستخدام قواعد نحوية مشابهة فإننا في هذا المثال نفترض أن كلاً من id , num هما مفردتان تم تحديد تركيباتهما أثناء مرحلة تحليل المفردات حيث يمكن افتراض أن المفردة num هي سلسلة من الأرقام والمفردة id هي سلسلة من الحروف.

وقد تم إضافة الجزء الخاص بالتعريفات إلى القواعد النحوية والذي يمثلة القاعدة النحوية الآتية:

exp → let dec-list in exp

حيث تشمل تعريفات متتابعة من التعريفات يفصل بينهما الفاصلة ',' وكل تعريف يكون على الشكل id = exp كما المثال التالي :

let x = 2 + 1, y = 3 + 4 in x + y

ودلالة التعبير let في القاعدة النحوية أن التعريفات التي تلي المفردة let تعطي أسماءً للتعبيرات بحيث يتم عند ظهور تلك الأسماء في التعبير الذي يلي المفردة in للاستعاضة عن هذه الأسماء بقيمة التعبيرات التي تمثلها الأسماء. وبالطبع فإن قيمة التعبير let هو القيمة النهائية للتعبير الذي يلي المفردة in والذي يتم حسابه بعد استبدال كل اسم في التعريفات بقيمة التعبير الذي يمثله هذا الاسم وعند تطبيق ذلك على المثال السابق فإن الاسم x يمثل القيمة 3 بينما الإسم y يمثل القيمة 7 وبالتالي فإن قيمة التعبير let هي 10 .

ومن الدلالة التي تم استخلاصها للتعبير let نجد أن التعريفات التي يحتوي عليها تشبه تعريفات الثوابت وأن تعبير let نفسه يمثل الكتلة الخاصة بهذه اللغة وبالتالي فنحن نحتاج إلى وصف قواعد المدى والتفاعل بين التعريفات في تلك التعبيرات حيث يلاحظ أن القواعد النحوية السابقة تسمح بالتفرغ في تعبيرات let إلى أي مستوى مثل التعبير التالي :

let x = 2, y = 3 in

(let x = x + 1, y = (let z = 3 in x + y + z)

(in (x +y

(

والآن نقوم بوضع قواعد المدى الخاص بتلك التعبيرات حتى يمكن تحديد قيمة التعبير السابق ولنبدأ بعدم السماح بإعادة تعريف نفس الاسم خلال نفس التعبير بمعنى إن التعبير الآتي:

let x = 2, x = 3 in x + 1

يعتبر غير سليم ويعطي خطأ.

أما القاعدة الثانية فهي إذا كان أي اسم غير معرف في المستوى الذي يوجد به التعبير أو في المستويات الأعلى منه يتم إظهار خطأ مثل التعبير التالي:

let x = 2 in x + y

والقاعدة الثالثة هي أن المدى لكل تعريف في التعبير let يمتد إلى جسم التعبير وفقا لمبدأ التفرع الأقرب في هيكل الكتلة بمعنى أن قيمة التعبير الآتي :

(let x = 2 in (let x = 3 in x

هي 3 وليس 2 ذلك لأن x في التعبير let الداخلي تعتمد على التعريف x = 3 وليس التعريف x = 2 لأنه التعريف الأقرب له.

وفي النهاية نقوم بتحديد التفاعل بين التعريفات في قائمة التعريفات بنفس التعبير على أنها متتابعة (Sequential) بمعنى أن كل تعريف يمكن أن يستخدم التعريفات السابقة في حل الأسماء الذي يحتوي عليها التعبير الذي يمثله فمثلا في التعبير التالي :

(let x = 2, y = x + 1 in (let x = x + y, y = x + y in y

فإن y الأولى تكون قيمتها 3 و x الثانية تكون قيمتها 5 أما y الثانية فإن قيمتها 8 وبالتالي فإن القيمة النهائية للتعبير let السابق هي 8 والآن وبعد الانتهاء من وضع القواعد المتعلقة بالمدى والتفاعل بين التعريفات ندعوا القارئ إلى حساب قيمة التعبير let الأساسي ذو مستويات التفرع الثلاثة والذي تم تقديمة في الصفحة السابقة.

ونحن نريد الآن إعداد معادلات القواعد الدلالية التي سترتبط بالقواعد النحوية السابقة على أن يتم استخدام جدول الرموز في متابعة تعريفات التعبيرات let وذلك وفقا لقواعد المدى والتفاعل التي تم الانتهاء من وصفها. وللتبسيط سيتم استخدام جدول الرموز فقط في التحديد ما إذا كان التعبير سليماً أم يتضمن أخطاء. وسوف نقوم بحساب الخاصية المصنعة err والتي تكون قيمتها true إذا كان التعبير يحتوي على أخطاء و false إذا كان التعبير سليماً وفقاً للقواعد. وللقيام بذلك نحتاج إلى كل من الخاصية الموروثة nestlevel التي تحدد ما إذا كان التعريفان في نفس كتلة let وهذه الخاصية تكون قيمتها رقم صحيح غير سالب يعبر عن مستوى التفرع الحالي للكتلة let وتأخذ القيمة 0 قبل البدء في أي تفرع.

والخاصية Symtab سوف تحتاج إلى جميع العمليات الخاصة بجدول الرموز ولأن تلك العمليات سيتم استخدامها من خلال القواعد الدلالية لذلك سوف يتم التعبير عنها بشكل مبسط. فعملية الإضافة سوف تأخذ جدول الرموز كمعطيات على أن ترجع جدول الرموز بعد إضافة المعلومات الجديدة مع الإحتفاظ بالجدول الأصلي بدون تغيير. فالعملية insert(s, n, l)i سوف تعود بجدول رموز جديد يحتوي على جميع المعلومات التي في الجدول s مع إضافة الإسم n مع مستوي التفرع l مع عدم تغيير الجدول s . وبالتالي فليس هناك ضرورة لوجود عملية منفصلة للحذف من جدول الرموز ولاختبار وجود الإسم في جدول الرموز. وكذلك استرجاع مستوى التفرع المصاحب له إذا كان موجوداً في الجدول، فإن هناك عمليتان للقيام بذلك الأولى هي isin(s , n)i التي تقوم بإرجاع true إذا كان الإسم n

موجودا في الجدول s و false إذا كان غير موجود في جدول الرموز s أو ترجع -1 إذا كان غير موجود.

ومع هذه العمليات الخاصة بجدول الرموز نستطيع الآن كتابة معادلات القواعد الدلالية للخصائص الثلاثة symtab و nestlevel و err الخاصة بالتعبيرات لتصبح القواعد الخصائصية كما يلي:

حيث يلاحظ استخدام جدولين مختلفين للرموز مع كل من dec-list و decl الأول هو intab الذي يمثل جدول الرموز قبل التعريفات والثاني outtab الذي يمثل نفس الجدول السابق ولكن بعد إضافة التعريفات كما يلاحظ أيضا ظهور حالة خاصة لجدول الرموز هي errtab والتي تحدث عند وجود خطأ في التعريف نتيجة لمخالفة قواعد المدى.

. تحليل الدلالات للغة الافتراضية

إن اللغة الإفترضية التي تم تقديمها في الوحدة الأولى وتم وصف القواعد النحوية الخاصة بها في الوحدة الرابعة بسيطة في احتياجاتها الخاصة بالدلالات الثابتة وسوف تعكس وحدة تحليل الدلالات تلك البساطة فلا يوجد أي تعريفات واضحة في هذه اللغة ولا يوجد تسمية للثوابت أو أنواع البيانات كما لا يوجد إجراءات والأسماء تستخدم فقط مع المتغيرات التي يتم تعريفها ضمنيا من خلال الاستخدام حيث يتم معاملة جميع المتغيرات على أنها متغيرات صحيحة (Integer Variables) ولا يوجد تفرع في المدى. وبالتالي يكون لاسم المتغير نفس المعنى خلال البرنامج بالكامل ولذلك فإن جدول الرموز لا يحتاج لأن يتعامل مع أي معلومات بخصوص المدى.

هذا سوف نقسم مناقشتنا بخصوص تحليل دلالات تلك اللغة الافتراضية إلى جزأين الأول لمناقشة هيكل جدول الرموز والعمليات المرتبطة به والثاني لوصف عمليات محلل الدلالات نفسه شاملا بناء جدول الرموز.

1.4. جدول رموز اللغة الافتراضية

لتصميم جدول الرموز الخاص بتحليل دلالات اللغة الافتراضية يجب أولا تحديد المعلومات التي نحتاج الاحتفاظ بها في هذا الجدول. وبصفة أساسية فإن هذه المعلومات تشمل نوع البيانات ومعلومات المدى. وحيث أن اللغة الافتراضية ليس لديها معلومات عن المدى كما أن جميع المتغيرات من نوع البيانات الصحيحة لذلك فإن جدول الرموز لا يحتاج أن يحتوي على مثل هذه المعلومات. ولكن أثناء مرحلة توليد شفرة البرنامج المستهدف من الترجمة فإن تلك المتغيرات التي يحتوي عليها البرنامج المطلوب ترجمته تحتاج إلى تخصيص موقع لها في الذاكرة ولأنه لا يوجد تعريف لهذه المتغيرات فإن جدول الرموز يكون هو المكان المنطقي للاحتفاظ بهذا الموقع المخصص للمتغيرات في الذاكرة وسوف نفترض الآن أن هذا الموقع سيتم تمثيله بعداد صحيح (Integer Counter) يتم زيادته مع كل متغير جديد ولكي يكون جدول الرموز أكثر فائدة سوف نستخدم الجدول أيضا في توليد قائمة بأرقام السطور الذي يظهر فيها كل متغير في البرنامج. وعلى سبيل المثال، فإن المعلومات التي سيتم توليدها بواسطة جدول الرموز بافتراض البرنامج التالي المكتوب باللغة الإفتراضية وهو نفس البرنامج السابق تقديمه بهذه اللغة من قبل ولكن بعد إضافة أرقام السطور إليه :

sample program} :1

computes factorial :2

{ :3

;read x :4

if 0 < x then :5

;fact := 1 :6

repeat :7

;fact := fact * x :8

x = x - 1 :9

;until x = 0 :10

write fact :11

end :12

وبعد إنشاء جدول الرموز للبرنامج السابق فإنه سيكون كما يلي:

مع ملاحظة أن ظهور المتغير أكثر من مرة في نفس السطر في البرنامج سيكرر رقم السطر أكثر من مرة مع هذا المتغير في جدول الرموز. وعموما فإن الهيكل المناسب لجدول الرموز في هذه الحالة هو هيكل الجدول العشوائي (Hash Table) ولأنه لا يوجد معلومات في الجدول عن المدى فليس هناك ضرورة لعملية الحذف من الجدول. أما عملية الإضافة فهي تحتاج فقط إلى رقم السطر وموقع الذاكرة كمعطيات بالإضافة إلى اسم المتغير وأخيرا عملية البحث تقوم بالبحث عن موقع اسم المتغير بالجدول.

2.4. محلل دلالات اللغة الافتراضية

إن الدلالات الثابتة للغة الافتراضية تشترك مع لغات البرمجة التقليدية في كون جدول الرموز أحد الخصائص الموروثة، بينما نوع البيانات الخاصة بأي تعبير هو خاصية مصنعة. وفي هذا الصدد يجب الإشارة إلى أن هذه اللغة يوجد بها أيضا نوع بيانات آخر بخلاف البيانات الصحيحة وهو نوع البيانات المنطقية (Boolean) ولكن هذا النوع الأخير من البيانات يظهر فقط كنتيجة لمقارنة قيمتين صحيحتين لأنه لا يوجد عمليات أو متغيرات من هذا النوع في اللغة. ويظهر التعبير المنطقي في الشرط الخاص

بأمر if أو أمر repeat ولا يمكن أن يظهر أي قيمة من هذا النوع في أي عملية كما أنه لا يمكن أن تمثل مخرجات باستخدام أمر write .

وعموما فإن جدول الرموز في هذه اللغة الافتراضية يمكن أن يتم بناءه بواسطة التنقل بالترتيب السابق (Preorder Traversal) لشجرة النحو.